异步图书
www.epubit.com

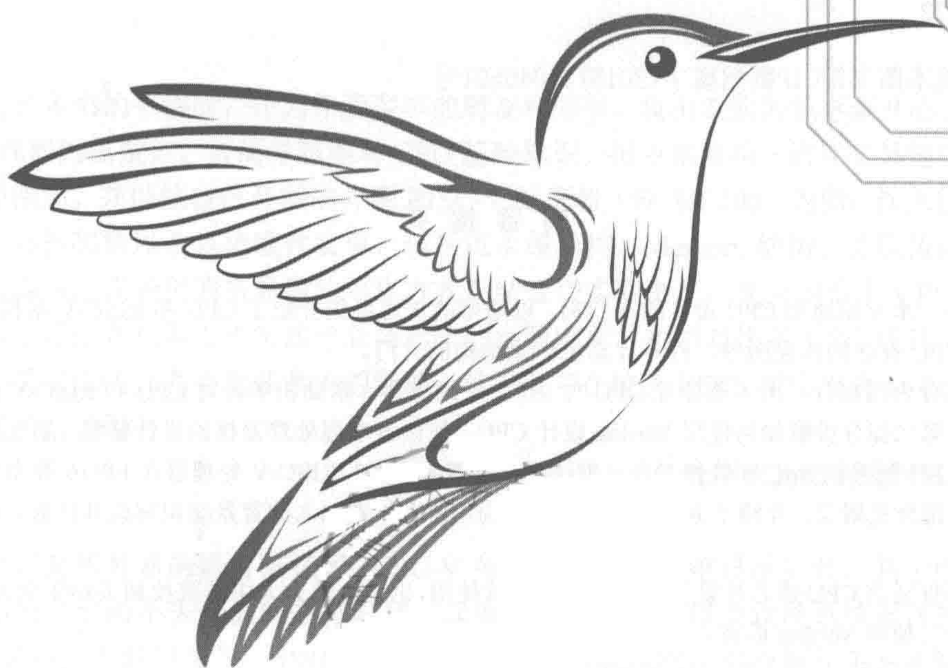
- ★ 深入浅出，介绍 CPU 设计技巧和新兴 RISC-V 指令集架构。
- ★ 内容全面，讲解开源蜂鸟 E200 系列处理器代码与设计精髓。
- ★ 画龙点睛，涵盖全套 SoC、软件工具链和 FPGA 原型平台的搭建和使用。



手把手教你设计 CPU——RISC-V 处理器篇

胡振波◎著

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS



手把手

教你设计CPU——RISC-V处理器篇

胡振波◎著

人民邮电出版社

北京

图书在版编目（C I P）数据

手把手教你设计CPU. RISC-V处理器篇 / 胡振波著
. -- 北京 : 人民邮电出版社, 2018.6
ISBN 978-7-115-48052-1

Ⅰ. ①手… Ⅱ. ①胡… Ⅲ. ①微处理器－系统设计
Ⅳ. ①TP332

中国版本图书馆CIP数据核字(2018)第046501号

内 容 提 要

本书是一本介绍通用 CPU 设计的入门书，以通俗的语言系统介绍了 CPU 和 RISC-V 架构，力求为读者揭开 CPU 设计的神秘面纱，打开计算机体系结构的大门。

本书共分为四部分。第一部分是 CPU 与 RISC-V 的综述，帮助初学者对 CPU 和 RISC-V 快速地建立起认识。第二部分讲解如何使用 Verilog 设计 CPU，使读者掌握处理器核的设计精髓。第三部分主要介绍蜂鸟 E203 配套的 SoC 和软件平台，使读者实现蜂鸟 E203 RISC-V 处理器在 FPGA 原型平台上的运行。第四部分是附录，介绍了 RISC-V 指令集架构，辅以作者加入的背景知识解读和注解，以便于读者理解。

本书不仅适合 CPU 或芯片设计相关从业者阅读使用，也适合作为大中专院校相关师生学习 RISC-V 处理器设计（使用 Verilog 语言）和 CPU 设计的指导用书。

◆ 著　　　　胡振波
　责任编辑　张　爽
　责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
　邮编　100164　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京市艺辉印刷有限公司印刷

◆ 开本：800×1000　1/16
　印张：26.75
　字数：598 千字　　　　2018 年 6 月第 1 版
　印数：1 – 3 000 册　　2018 年 6 月北京第 1 次印刷

定价：99.00 元

读者服务热线：(010) 81055410　印装质量热线：(010) 81055316
反盗版热线：(010) 81055315
广告经营许可证：京东工商广登字 20170147 号

精彩书评

当拿到这本书的书稿时，作为作者多年的朋友和同事，我由衷地为他感到开心。作者以RISC-V处理器为出发点，言简意赅地对CPU基础知识、指令集架构，软件工具链等核心技术做了原理阐述，并以他自己开发的开源RISC-V处理器（蜂鸟E200）为例，深入浅出地介绍了该处理器微架构以及具体硬件实现，包括流水线结构、Memory结构、总线协议、中断异常和调试器等。作者用通俗易懂的写作方式，结合切实的例子，充分剖析了CPU设计的全过程。本书是作者长期工作实践中总结出来的经验宝典，相信对很多CPU设计爱好者以及想从事相关工作的读者会有非常大的帮助。对于很多在校的EECS学生来说，这也将是一本不可多得的指导用书。

——Synopsys公司 ARC处理器高级研发经理 彭剑英

这本书以简练朴素的语言描述了RISC-V处理器架构的完整设计实践，其中既有对处理器体系结构简练而不失全面的总结对比，又有RISC-V处理器及软硬系统设计的诸多细节。“小小麻雀，五脏俱全”，书中内容反映着作者对本领域的深刻理解和亲身实验，值得学习借鉴。

初识本书作者是在处理器设计架构的研讨会中，他勤勉务实的作风给人留下了深刻的印象，我想也影响着本书的行文和风格，值得学习。

——国防科技大学 教授 姜晶菲

RISC-V处理器已经在工业界大放异彩，预计不久之后在“微机原理”和“计算机体系结构”课程中讲解RISC-V将像在“操作系统”课程中讲解Linux一样成为主流。本书作者领导的团队开源了“蜂鸟E203”处理器，并提供了完整的开发环境，为读者搭建了从RISC-V理论学习过渡到工程实践的桥梁。本书不仅深入地诠释了项目代码，而且凝聚了作者多年从事CPU设计工作的经验与感悟。对于电子类和计算机类专业的师生、CPU技术爱好者和从业者来说，本书极具参考价值！

——天津大学电气自动化与信息工程学院 副教授 吕卫

本书介绍的蜂鸟E203 RISC-V处理器核，“蜂鸟虽小，五脏俱全”。本书不仅适合作为大

中专院校师生学习 RISC-V 处理器设计（使用 Verilog 语言）的教学或自学案例，而且在 IoT 领域也会大有可为。结合该处理器实例与作者多年的 CPU 设计经验与心得，本书用通俗易懂的语言，为读者轻松揭开了 CPU 设计的神秘面纱，非常适合作为大中专院校师生学习 RISC-V 和 CPU 设计的教学书籍。

——华中科技大学微电子工程系 副教授 郑朝霞

本书非常全面地介绍了 RISC-V 开发所需的知识点，内容丰富，实用性非常强，并且详尽地介绍了 RISC-V 的架构设计和性能优化方面的内容，必将成为 RISC-V 开发者的必备。我打算用本书和蜂鸟 E200 作为学生的课程学习资料，相信大家会有所收获。

——西安邮电大学 高工、博士 焦继业

长期以来，由于商用处理器 IP 核高昂的授权费用和商业保密因素的影响，商用处理器微体系结构总是“秘不示人”。这导致大量学习者只能从框图和体系结构仿真软件上去学习“概念化”的处理器微体系结构，与实战差距较大。而本书作者研发的蜂鸟 E200 开源处理器是中国较早的基于 RISC-V 指令集开发的开源处理器。在本书中，作者以蜂鸟 E200 处理器为例，介绍了从处理器微体系结构到片上系统设计的相关知识。更为可贵的是，作者是以“集成电路设计视角”，而非“计算机系统结构视角”来介绍这些知识的，这样的知识组织结构更加符合集成电路设计从业人员的学习习惯和思考方式。因此，这本书对于从事数字集成电路设计的工程师、高校教师以及相关专业的学生是一本难得的参考书籍。

——电子科技大学电子科学与工程学院 副教授 黄乐天

有幸同本书作者共事过一年，每每被他的知识面之广博所折服。今日读到此书，更加佩服！RISC-V 在需要低功耗和可配置性的场合有强大的生命力，在 IoT 领域以及高性能定制（例如 AI 加速）芯片领域即将大放异彩，此书来得非常及时！如果您是硬件背景的工程师，此书可以让您快速上手 RISC-V，增强自身的竞争力。如果您是软件背景的工程师，此书轻松易懂，可以替代 *Computer Systems: A Programmer's Perspective*（《深入理解计算机系统》）一书，让您了解计算机系统的底层是如何工作的。

——北京比特大陆 资深芯片和 CPU 设计专家 王逵

2018 年，在 meltdown 和 spectre 两大芯片设计的漏洞爆发之后，我愈加感觉到一个程序员了解和理解处理器的设计原理和工作机制是多么的必要。这是一本能让你学习到上述知识的好书，所有程序员都应该读一读。

——《奔跑吧 Linux 内核》作者 笨叔叔

序 一

芯片，是整个电子信息产业的基石。目前，全球半导体市场规模达 3200 亿美元，全球 54%的芯片都出口到了中国，但国产芯片的市场份额只占 10%。中国芯片产业每年进口需要消耗 2000 多亿美元外汇，超过了石油和大宗商品，在进口商品中占有相当大的比重。

CPU 作为芯片的“心脏”，可谓“芯中之心”，国内的产业实力在此方面一直比较薄弱。CPU 实现国产自主化对我国的发展至关重要，但是 CPU 的主流指令集架构（譬如 x86 和 ARM）一直为国外公司所垄断，国内公司需要支付高昂的专利费用且受制于人。CPU 作为一种特殊的芯片，其要求指令集架构具有普世的通用性且能够共享生态系统，因此囿于一国范围内发明一套封闭的指令集并不具备实用性，必须走与世界主流架构接轨的道路。在这种背景下，开放的 RISC-V 架构给中国 CPU 芯片产业的发展带来了巨大的战略机遇，有希望彻底实现 CPU 的国产自主化和架构主流化。

目前，我国正处于大力发展芯片设计行业的关键时期，实现中华民族伟大复兴的重任需要广大科研和工程工作者孜孜不倦地努力与拼搏，需要很多像作者这样求真务实的技术中坚力量来担负起国产芯片振兴的重任。而国内 CPU 领域人才的奇缺是长期制约行业发展的主要因素，本书作者作为一名长期工作在一线的资深 CPU 设计专家，将其经验撰写成书，资料翔实，文字生动。配合作者所在公司开发的蜂鸟 E200 系列处理器核作为实例，非常适合用于教学领域以及爱好者学习，对于普及 CPU 的设计技术具有十分积极的意义。

新兴的 RISC-V 架构在全球范围内已经掀起了一场热潮，在国内也引起了广泛的关注，但是由于没有很好的中文普及书籍，很多人对于 RISC-V 仍然是“只闻其声，未见其形”。作者作为国内第一批接触 RISC-V 架构，并最早研发成功 RISC-V 处理器的技术专家，在工作之余将其自研的处理器核开源，并著书详细解读其实现细节，体现了作者极高的专业水准和推进国产 CPU 产业发展的炽热情怀。

本书以极为通俗易懂的语言对 RISC-V 架构进行了系统而全面地介绍，并且结合蜂鸟 E200 系列开源处理器核对 CPU 设计技术进行了深入浅出的讲解，图文并茂，生动活泼，体现了作者深厚的专业技能以及将专业知识进行通俗化表述的优秀能力。令人印象深刻的是，本书作者在对 RISC-V 架构进行介绍的过程中，加入了大量的背景知识解读以及个人注解，使得枯燥的专业知识变得非常易于理解，可以说是难能可贵。这是一本凝聚了作者多年所学的精心之作，非常值得一读，对于 RISC-V 架构在国内的传播也将具有巨大的推动作用。本书作为国内不可多得的介绍 RISC-V 的中文书籍，相信一定会成为该领域的经典之作。

邹雪城
华中科技大学武汉国际微电子学院执行院长、教授
武汉集成电路设计工程技术研究中心主任
2018 年初 于武汉

序 二

近 40 年来，通用计算的两次跨时代的飞跃都来自硬件系统的逐步开放。以 PC 为终端的有线互联网技术完成了地址与地址之间的通信，让数据真正实现了大量传输与交互。而这一切源于美国 Intel 公司发明 x86 芯片，并将芯片提供给各大计算机制造厂商按照开放硬件标准组装，使 PC 大批量制造成为可能。PC 的普及开创了计算机互联网的时代，互联网时代开始之初的用户数量是 5 亿（2003 年）。以手机为终端的移动互联网实现了人与人之间的通信，让终端个人直接连接到网络。这一切都是因为英国 ARM 公司将 CPU 的 IP 授权给手机芯片厂商以 SoC 片上系统的形式研制自己的芯片，使智能手机得以大量普及，同时开创了移动互联网时代，移动互联网时代开始之初的用户数量是 25 亿（2009 年）。

未来是物联网的时代，那时的用户数量会是多少呢？有分析称，2020 年将会达到 300 亿，2050 年极保守估算将是 1000 亿以上。美国公司通过完成硬件开放标准化，研发芯片并将其出售，完成了 5 亿用户普及；英国公司通过完成芯片标准化，并将其芯片 IP 有限的授权，完成了 25 亿用户的推广。而面对一个上百亿用户推广的目标，数以千万计的不同应用需求，必须要有更开放的形式吸引更多的力量参与，而这个答案显而易见就是——开源。只有开源，才能降低进入门槛，加快技术迭代速度，实现技术普及，满足市场需求。开源、共享、共赢，可以说在物联网时代，开源的硬件是一种必需，没有基础芯片技术的进一步开源，就没有物联网应用的未来。

而且，在摩尔定律减缓的今天，一味比拼硬件性能的技术竞赛变得越发艰难。然而性能提升的最终目的是满足应用，如何在现有的能力下最大程度地满足数以千万计的不同应用的需求，就成了当今处理器行业要面对的重要问题。RISC-V 站在了时代的风口，其作为一种开源架构的出现必将对芯片产业产生深远的影响。硬件芯片的开源不再是一个噱头，而是变成了一种刚需。蜂鸟 E200 作为中国本土较早开源的 RISC-V 芯片，也将对国内的相关产业发展起到巨大的推动作用，为国内公司抓住物联网风口的大发展而助力。开源即透明，透明即可控。自主可控也是国家信息安全的保障，由国内公司自主研发并开源的 RISC-V 处理器，也为国家信息安全领域的应用提供了多一种选择。

这本书基于蜂鸟 E200，全面而详细地介绍了 RISC-V 处理器，分四大部分讲解了处理器和指令集架构的基本知识与背景、处理器设计要诀、软件运行实例以及 RISC-V 附录资料，是国内 RISC-V 开源处理器社区极其稀缺的教材，可以说是 RISC-V 芯片版的《鸟哥的 Linux 私房菜》。更难能可贵的是，书中作者不但深入浅出地讲解了处理器相关的几乎全部的基本

概念和功能设计要点，而且辅以大量实例与解析，是初学者在 RISC-V 和处理器设计领域推门入室的不二之选。

我推荐这本书，并不仅仅因为它是一本好的技术教材，更是因为它可以为更多人开启开源硬件、芯片世界的大门。开源的本质是技术合作和知识的自由传播。知识只有传播才有价值，而开源就是让知识能在广泛群体中传播，并凝聚这个群体的力量来推动相应技术的发展。开源精神可能是一种理想主义，而这种理想主义也是有其现实意义的，尤其是在著作权和专利构成技术进步壁垒下的今天，创新需要这样的理想主义来推动，而理想主义的世界需要优秀的作品来开启。

期待这本优秀的教材能为读者打开理想主义世界的大门！

郑云龙
中国科学院博士后
九天微星技术总监
2018 年初 于北京

前 言

永恒的热点——CPU

灯，等灯等灯……

——Intel

如果要评选过去十数年间经典的科技广告音乐，想必 Intel 的广告音乐“灯，等灯等灯……”会榜上有名。而熟悉的蓝色贴标“Intel Inside”，也必是辨识度很高的广告图标。

中央处理单元（Center Processor Unit，CPU），虽然早已被大众所熟知，但在相当长的时间内，它一直是高端大气的代名词。也许很多读者都和作者有着相同的印象，在早期的电脑杂志上，必然有着浓墨重彩的篇幅与专栏，详细介绍 Intel 或 AMD 公司推出的 CPU 芯片的详细参数，抑或 MIPS 与 ARM 的优劣之处，甚至是令人津津乐道的各家 CPU 的小道消息与幕后桥段。

而 CPU 代表的高端技术，也一直笼罩在神秘的面纱之下。当作者还是一名学生的时候，每当浏览 CPU 的论坛与探讨文章，总能看到其作为热门版块吸引着众多的拥护者，并拥有极高的下载量。当我结束了多年的校园生活，开始选择人生第一份工作时，正逢某知名外企到中国设立 CPU 研发部门。经过激烈的竞争，我有幸成为了一名 CPU 逻辑设计工程师。而后多年间，我先后供职于多家国际一流公司的 CPU 设计部门，参与过多次校园招聘和社会招聘，CPU 设计部门总能以其光环吸引众多的应聘者前来竞争角逐。就像每个男孩心中都有一个军人梦一样，似乎每一个芯片设计人员都有一个 CPU 梦。

当今世界，科学技术正在以令人惊异的速度飞快发展，从 IoT（物联网）到大数据再到人工智能，新技术与新领域如雨后春笋般层出不穷。而 CPU 这门诞生于 20 世纪 60～70 年代的技术，是否已经垂垂老矣，应该退出历史舞台了呢？其实不然，从 IoT 的超低功耗微控制器到大数据的高性能计算，再到人工智能的异构计算，CPU 均扮演着核心的角色。可以说，在未来相当长的时间内，从指尖的超低功耗处理器到云端的超级计算机，CPU 技术都将持续站在前沿科技的风口浪尖，散发着它历久弥新的魅力。

然而长期以来，CPU 架构主要由以 Intel（x86 架构）和 ARM（ARM 架构）为代表的商业巨头公司所掌控，成为普通公司与个人无法逾越的天堑。2016 年，RISC-V 基金会成立，开放免费的 RISC-V 架构。这一举动具有划时代的意义，任何公司与个人均可依据开放的 RISC-V 架构设计自己所需的处理器，极大地降低了 CPU 的准入门槛。因此，RISC-V 架构在极短的时间内便引起了业界的高度关注，从众多反应快速的小公司到实力雄厚的巨头公司（如 NVIDIA、三星等）均开始使用 RISC-V 架构开发产品。“旧时王谢堂前燕，飞入寻常百姓家”，在摩尔定

律逐步逼近极限的今天，开放且免费的 RISC-V 架构的诞生，将催生出新一轮的创新热潮。

当前国内 CPU 产业方兴未艾，x86、ARM 和 MIPS 等传统商用处理器架构呈现“百花齐放”之势。龙芯、兆芯、飞腾等资深专业 CPU 公司在不断突破，华为、展讯等一线大公司相继开始研发自主的处理器核心，海光、华芯通等新锐也开始摩拳擦掌。此时，开放的 RISC-V 架构的诞生，更是锦上添花。可以说，学习 CPU 设计正当时，学习 RISC-V 正当时！

作者在培训新入门工程师，或与爱好者、学生交流的过程中，能够感到 CPU 设计这门要求计算机体系结构和软硬件皆通的技术令初学者难以学习和掌握，作者时常遗憾于没有很好的通俗读本。正所谓“曲高和寡，妙伎难工”，CPU 设计过于专业，相关的书籍或卷帙浩繁，或晦涩难懂，令初学者不知从何下手，且难以理解。在实际的练习中，也难以找到易于学习和上手的例程，更别说完整地设计一款处理器了。

如上种种，正是促使作者撰写此书的原因。作者希望本书能够作为一本通俗读本，帮助初学者和爱好者顺利越过初期的陡峭学习曲线，进入 CPU 设计的坦途。谨以此书献给曾经帮助过作者的良师益友、合作伙伴和默默工作的工程师们！

本书内容

Stay hungry, Stay foolish.
（求知似饥，虚心若愚。）
—— Steven Jobs（史蒂夫·乔布斯）

- 您是否想学习工业级 Verilog RTL 数字 IC 设计的精髓与技巧？
- 您是否阅读了众多计算机体系结构的书籍仍不明就里？
- 您是否想揭开 CPU 设计神秘的面纱，并亲自设计一款处理器？
- 您是否想学习国际一流公司真实的 CPU 设计案例？
- 您是否想用最短的时间熟悉并掌握 RISC-V 架构？
- 您是否想深入理解并使用一款免费可靠的开源 RISC-V 处理器和完整的 SoC 平台？

如果您对上述任意一个问题感兴趣，本书都将是您很好的选择。

作者所在公司的团队，总结各国际一流公司多年从事 CPU 设计工作的丰富经验，开发了一款超低功耗 RISC-V 处理器（蜂鸟 E200），也是一款开源的 RISC-V 处理器。

结合该处理器实例与作者多年的 CPU 设计经验与心得，本书将用通俗易懂的语言，深入浅出地剖析 RISC-V 处理器的微架构以及代码实现，为读者揭开 CPU 设计的神秘面纱，打开计算机体系结构的大门。

本书旨在成为国内第一本系统介绍 RISC-V 指令集架构的通俗读本，以及第一本结合实际 RISC-V 开源实例进行教学的技术图书。相信通过对本书的学习，读者能够快速掌握并轻松使用 RISC-V 架构处理器。

通过学习实例蜂鸟 E200 的 Verilog 代码，您将能成为一名合格的数字 IC 设计工程师；通过

学习本书推荐的完整开源 SoC 平台，您也可以快速搭建 FPGA 原型平台，运行完整的软件实例。

希望本书能够为科普 RISC-V 指令集架构起到推动作用，同时通过对“蜂鸟 E200 处理器”的开源与解析，为 RISC-V 处理器在国内的普及贡献绵薄之力。本书共分四部分，各部分主要内容如下。

第一部分是 CPU 与 RISC-V 综述，包括第 1～4 章。该部分将介绍 CPU 的一些基础背景知识、RISC-V 架构的诞生和特点。

第 1 章主要介绍 CPU 的基础知识、指令集架构的历史、国产 CPU 的发展现状及原因、CPU 的应用领域、各领域的主流架构、RISC-V 的诞生背景等。

第 2 章主要介绍 RISC-V 架构和特点，着重分析其大道至简的设计哲学，并阐述 RISC-V 和以往曾经出现过的开放架构有何不同。

第 3 章主要对当前全球范围内的商业或者开源 RISC-V 处理器进行盘点，分析其优缺点，并引出开源的 RISC-V 处理器——蜂鸟 E200 系列处理器核和 SoC。

第 4 章主要对蜂鸟 E200 系列处理器核和 SoC 的特性进行整体介绍。

第二部分主要讲解如何使用 Verilog 设计 CPU，包括第 5～16 章。该部分将对蜂鸟 E200 处理器核的微架构和源代码进行深度剖析，结合该处理器核进行处理器设计案例分析。

第 5 章主要先从宏观的角度着手，介绍若干处理器设计的技巧、蜂鸟 E200 处理器核的总体设计思想和顶层接口。帮助读者整体认识蜂鸟 E200 处理器的设计要诀，为后续各章针对不同部分展开详述奠定基础。

第 6 章主要介绍处理器的一些常见流水线结构，并介绍蜂鸟 E200 处理器核的流水线结构。

第 7 章主要介绍处理器的取指功能，并介绍蜂鸟 E200 处理器核取指单元的微架构和源码分析。

第 8 章主要介绍处理器的执行功能，并介绍蜂鸟 E200 处理器核执行单元的微架构和源码分析。

第 9 章主要介绍处理器交付的功能和常见策略，并介绍蜂鸟 E200 处理器核交付单元的微架构和源码分析。

第 10 章主要介绍处理器的写回功能和常见策略，并介绍蜂鸟 E200 处理器核的写回硬件实现微架构和源码分析。

第 11 章主要介绍处理器的存储器架构，并介绍蜂鸟 E200 处理器核存储器子系统的微架构和源码分析。

第 12 章主要介绍蜂鸟 E200 处理器核的总线接口模块，介绍其使用的总线协议，以及该模块的微架构和源码分析。

第 13 章主要介绍 RISC-V 架构定义的中断和异常机制，蜂鸟 E200 处理器核中断和异常的硬件微架构及其源码分析。

第 14 章主要介绍处理器的调试机制，介绍 RISC-V 架构定义的调试方案、蜂鸟 E200 处理器调试机制的硬件实现微架构和源码分析。

第 15 章主要介绍处理器的低功耗技术，并以蜂鸟 E200 处理器为例阐述其低功耗设计的诀窍。

第 16 章主要介绍如何利用 RISC-V 的可扩展性，并以蜂鸟 E200 的协处理器接口为例详细阐述如何定制一款协处理器。

第三部分是使用 Verilog 进行仿真和在 FPGA SoC 原型上运行软件，包括第 17～20 章。该部分将对蜂鸟 E200 配套 SoC 的软硬件平台进行剖析，并详细讲解如何进行 Verilog 仿真测试，如何在 FPGA 原型上运行软件示例程序和使用 GDB 进行调试。想快速使用蜂鸟 E200 的读者可以跳过第二部分，直接阅读第三部分内容。

第 17 章主要介绍蜂鸟 E200 开源平台如何运行 Verilog 仿真测试。

第 18 章主要介绍蜂鸟 E200 处理器配套的 SoC 如何在 FPGA 上实现该 SoC 原型。

第 19 章主要介绍如何使用 SoC 的 FPGA 原型平台运行真正的软件示例、如何使用 GDB 对程序进行调试。

第 20 章主要介绍如何使用 SoC 的 FPGA 原型平台运行跑分程序，对蜂鸟 E200 处理器核的性能进行量化评估。

第四部分是附录部分，包括附录 A～附录 G。该部分将对 RISC-V 指令集架构进行详细介绍，对 RISC-V 指令集架构细节感兴趣的读者可以先行阅读附录部分。

附录 A 主要介绍 RISC-V 架构的指令集。该附录翻译自 RISC-V 的“指令集文档”，并对相关内容进行了重新组织，以求通俗易懂。

附录 B 主要介绍 RISC-V 架构的 CSR 寄存器。该附录对于 CSR 寄存器的介绍翻译自 RISC-V 的“特权架构文档”，同时还介绍了蜂鸟 E200 处理器核自定义的 CSR 寄存器。

附录 C 主要介绍 RISC-V 架构定义的系统平台中断控制器（Platform Level Interrupt Controller，PLIC）。该附录对于 PLIC 的介绍翻译自 RISC-V 的“特权架构文档”。

附录 D 主要介绍存储器模型（Memory Model）的相关背景知识，帮助读者更深入地理解 RISC-V 架构的存储器模型。

附录 E 主要结合多线程“锁”的示例对存储器原子操作指令的应用背景进行简介。

附录 F 和附录 G 分别是 RISC-V 指令的编码列表和 RISC-V 伪指令的列表。附录均节取自 RISC-V 的“指令集文档”，供读者快速查阅。

建议与反馈

由于时间仓促且作者水平有限，书中难免存在不足之处，敬请各位读者批评指正。相关问题请与本书编辑（zhangshuang@ptpress.com.cn）联系交流，或到异步社区（https://www.epubit.com/）中提交勘误。

资源与支持

本书为异步社区出品的图书，在社区（https://www.epubit.com/）上为您提供以下服务。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免还会存在差错。欢迎您将发现的问题告诉我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区主页 https://www.epubit.com/，搜索到本书页面，点击“提交勘误”，输入相应信息，最后点击“提交”按钮即可。之后本书的作者和编辑会对您提交的勘误进行审核。确认并接受后，您将获赠异步社区的 100 积分。积分可用于在社区兑换优惠券，以及兑换样书或奖品之用。

扫码关注本书

请扫描下方二维码关注本书，即可在异步社区微信服务号中看到本书和进一步的服务信息。



与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请发邮件到此邮箱，邮件标题中请注明本书书名。

如果您有兴趣出版图书、录制教学视频，或参与图书翻译、技术审校等工作，可以发邮件，或者到异步社区在线提交投稿：

https://www.epubit.com/selfpublish/submission

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，请发邮件联系我们。

如果您在网上发现有针对异步图书的各种形式的盗版行为，包括图书或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的举动是对作者权益的保护，我们也由此才能继续为您带来有价值的内容。

关于异步社区和异步图书

异步社区是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书和相关学习产品，为作译者提供优质出版服务，社区创办于 2015 年 8 月，提供超过 1000 种图书、近 1000 种电子书，以及众多技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

异步图书是由异步社区编辑团队策划出版的精品 IT 专业图书品牌，依托于人民邮电出版社近 30 年的计算机图书出版积累和专业编辑团队，在封面上印有异步图书的 LOGO。我们的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。



目录

第一部分 CPU 与 RISC-V 综述

第二部分 手把手教你使用 Verilog 设计 CPU

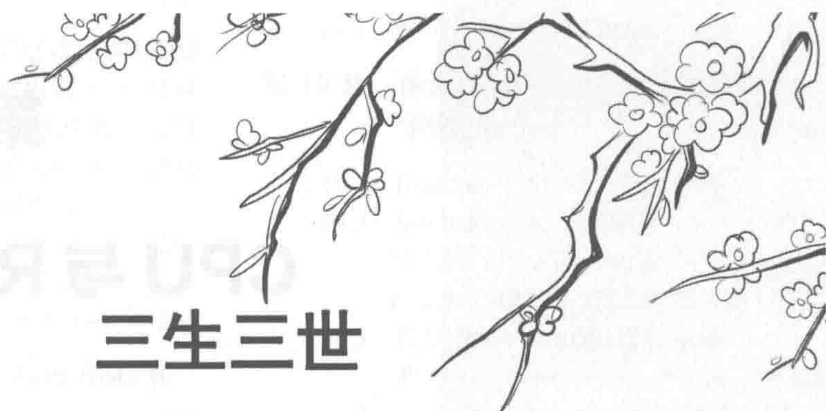
第三部分 使用 Verilog 进行仿真和在 FPGA SoC 原型上运行软件

附录部分 RISC-V 架构详述

第一部分

CPU与RISC-V综述

第 1 章　一文读懂 CPU 之三生三世



三生三世



本章通过几个轻松的话题，讨论一下 CPU 业界的“三生三世”。

1.1 眼看他起高楼，眼看他宴宾客，眼看他楼塌了——CPU 众生相

CPU，全称为中央处理器单元，简称为处理器，是一个不算年轻的概念。早在 20 世纪 60 年代便已诞生了第一款 CPU。

请注意区分“处理器”和“处理器核”“CPU”和“Core”的概念。严格来说，“处理器核”和“Core”是指处理器内部最核心的部分，是真正的处理器内核；而“处理器”和“CPU”往往是一个完整的 SoC，包含了处理器内核和其他的设备或者存储器。但是在现实中大多数文章往往不会严格地遵循两者的差别，时常混用，因此读者需要根据上下文自行甄别体会具体的含义。

经过几十年的发展，到今天已经相继诞生或消亡过了几十种不同的 CPU 架构。表 1-1 为近几十年来知名 CPU 架构的诞生时间表。什么是 CPU 架构？下面让我们来探讨区分 CPU 的主要标准：CPU 的灵魂——指令集架构（Instruction Set Architecture，ISA）。

表 1-1　知名 CPU 架构的诞生时间表

CPU 架构	诞生时间/年
IBM 701	1953
CDC 6600	1963
IBM 360	1964
DEC PDP-8	1965
Intel 8008	1972
Motorola 6800	1974
DEC VAX	1977
Intel 8086	1978
Intel 80386	1985
ARM	1985
MIPS	1985
SPARC	1987
Power	1992
Alpha	1992
HP/Intel IA-64	2001
AMD64 (EMT64)	2003

1.1.1 ISA——CPU 的灵魂

指令集，顾名思义是一组指令的集合，而指令是指处理器进行操作的最小单元（譬如加减乘除操作或者读/写存储器数据）。

指令集架构，有时简称为“架构”或者称为“处理器架构”。有了指令集架构，便可以使用不同的处理器硬件实现方案来设计不同性能的处理器。处理器的具体硬件实现方案称为微架构（Microarchitecture）。虽然不同的微架构实现可能造成性能与成本的差异，但是，软件无须做任何修改便可以完全运行在任何一款遵循同一指令集架构实现的处理器上。因此，指令集架构可以理解为一个抽象层，如图 1-1 所示。该抽象层构成处理器底层硬件与运行于其上的软件之间的桥梁与接口，也是现在计算机处理器中重要的一个抽象层。

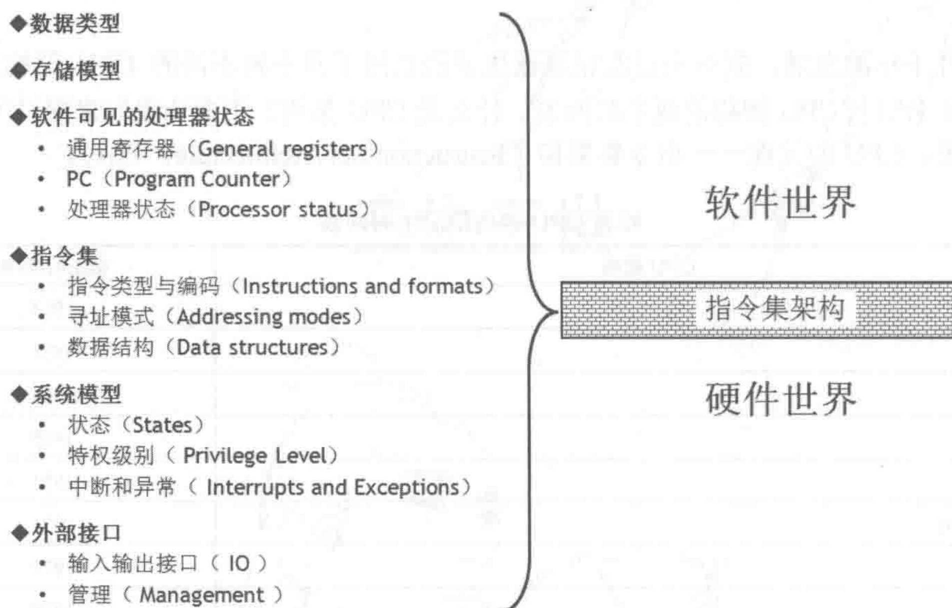
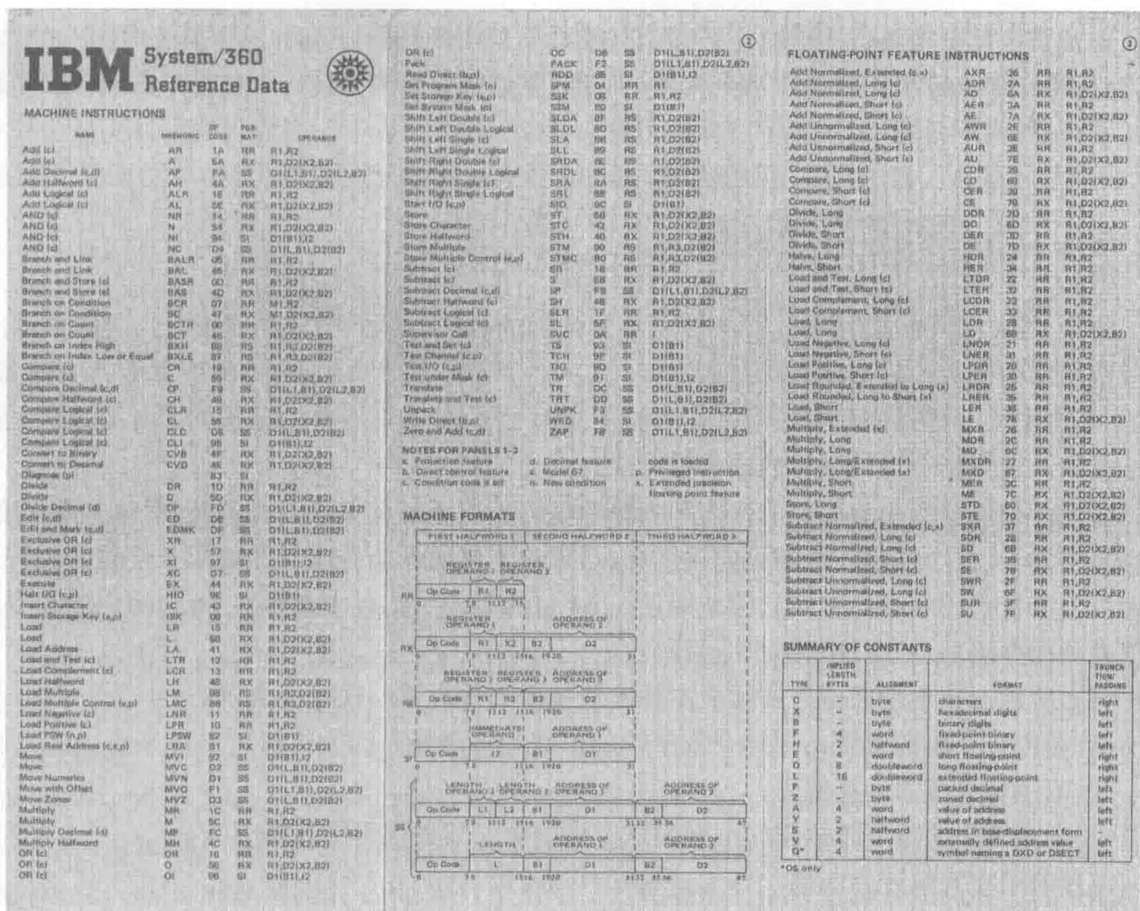


图 1-1 指令集架构示意图

为了让软件程序员能够编写底层的软件，指令集架构不仅仅是一组指令的集合，它还要定义任何软件程序员需要了解的硬件信息，包括支持的数据类型、存储器（Memory）、寄存器状态、寻址模式和存储器模型等。如图 1-2 所示，IBM 360 指令集架构是第一个里程碑式的指令集架构，它第一次实现了软件在不同 IBM 硬件机器上的可移植性。

综上可见，指令集架构才是区分不同 CPU 的主要标准，这也是 Intel 和 AMD 公司多年来分别推出了几十款不同的 CPU 芯片产品的原因。虽然来自于两个不同的公司，但是它们仍被统称为 x86 架构 CPU。



IBM System/360 Reference Data

MACHINE INSTRUCTIONS

NAME	MNEMONIC	OP CODE	FORMAT	OPERANDS
Add (c)	AR	1A	RR	R1,R2
Add (c)	A	5A	RX	R1,D2(X2,B2)
Add Decimal (c,d)	AP	FA	SS	D1(L1,B1),D2(L2,B2)
Add Halfword (c)	AH	4A	RX	R1,D2(X2,B2)
Add Logical (c)	ALR	1E	RR	R1,R2
Add Logical (c)	AL	5E	RX	R1,D2(X2,B2)
AND (c)	NR	14	RR	R1,R2
AND (c)	N	54	RX	R1,D2(X2,B2)
AND (c)	NI	94	SI	D1(B1),I2
AND (c)	NC	D4	SS	D1(L,B1),D2(B2)
Branch and Link	BALR	05	RR	R1,R2
Branch and Link	BAL	45	RX	R1,D2(X2,B2)
Branch and Store (e)	BASR	0D	RR	R1,R2
Branch and Store (e)	BAS	4D	RX	R1,D2(X2,B2)
Branch on Condition	BCR	07	RR	M1,R2
Branch on Condition	BC	47	RX	M1,D2(X2,B2)
Branch on Count	BCTR	06	RR	R1,R2
Branch on Count	BCT	46	RX	R1,D2(X2,B2)
Branch on Index High	BXH	86	RS	R1,R3,D2(B2)
Branch on Index Low or Equal	BXLE	87	RS	R1,R3,D2(B2)
Compare (c)	CR	19	RR	R1,R2
Compare (c)	C	59	RX	R1,D2(X2,B2)
Compare Decimal (c,d)	CP	F9	SS	D1(L1,B1),D2(L2,B2)
Compare Halfword (c)	CH	49	RX	R1,D2(X2,B2)
Compare Logical (c)	CLR	15	RR	R1,R2
Compare Logical (c)	CL	55	RX	R1,D2(X2,B2)
Compare Logical (c)	CLC	D5	SS	D1(L,B1),D2(B2)
Compare Logical (c)	CLI	95	SI	D1(B1),I2
Convert to Binary	CVB	4F	RX	R1,D2(X2,B2)
Convert to Decimal	CVD	4E	RX	R1,D2(X2,B2)
Diagnose (p)		83	SI	
Divide	DR	1D	RR	R1,R2
Divide	D	5D	RX	R1,D2(X2,B2)
Divide Decimal (d)	DP	FD	SS	D1(L1,B1),D2(L2,B2)
Edit (c,d)	ED	DE	SS	D1(L,B1),D2(B2)
Edit and Mark (c,d)	EDMK	DF	SS	D1(L,B1),D2(B2)
Exclusive OR (c)	XR	17	RR	R1,R2
Exclusive OR (c)	X	57	RX	R1,D2(X2,B2)
Exclusive OR (c)	XI	97	SI	D1(B1),I2
Exclusive OR (c)	XC	D7	SS	D1(L,B1),D2(B2)
Execute	EX	44	RX	R1,D2(X2,B2)
Halt I/O (c,p)	HIO	9E	SI	D1(B1)
Insert Character	IC	43	RX	R1,D2(X2,B2)
Insert Storage Key (a,p)	ISK	09	RR	R1,R2
Load	LR	18	RR	R1,R2
Load	L	58	RX	R1,D2(X2,B2)
Load Address	LA	41	RX	R1,D2(X2,B2)
Load and Test (c)	LTR	12	RR	R1,R2
Load Complement (c)	LCR	13	RR	R1,R2
Load Halfword	LH	48	RX	R1,D2(X2,B2)
Load Multiple	LM	98	RS	R1,R3,D2(B2)
Load Multiple Control (e,p)	LMC	B8	RS	R1,R3,D2(B2)
Load Negative (c)	LNR	11	RR	R1,R2
Load Positive (c)	LPR	10	RR	R1,R2
Load PSW (n,p)	LPSW	82	SI	D1(B1)
Load Real Address (c,e,p)	LRA	B1	RX	R1,D2(X2,B2)
Move	MVI	92	SI	D1(B1),I2
Move	MVC	D2	SS	D1(L,B1),D2(B2)
Move Numerics	MVN	D1	SS	D1(L,B1),D2(B2)
Move with Offset	MVO	F1	SS	D1(L1,B1),D2(L2,B2)
Move Zones	MVZ	D3	SS	D1(L,B1),D2(B2)
Multiply	MR	1C	RR	R1,R2
Multiply	M	5C	RX	R1,D2(X2,B2)
Multiply Decimal (d)	MP	FC	SS	D1(L1,B1),D2(L2,B2)
Multiply Halfword	MH	4C	RX	R1,D2(X2,B2)
OR (c)	OR	16	RR	R1,R2
OR (c)	O	56	RX	R1,D2(X2,B2)
OR (c)	OI	96	SI	D1(B1),I2
OR (c)	OC	D6	SS	D1(L,B1),D2(B2)
Pack	PACK	F2	SS	D1(L1,B1),D2(L2,B2)
Read Direct (b,p)	RDD	85	SI	D1(B1),I2
Set Program Mask (n)	SPM	04	RR	R1
Set Storage Key (a,p)	SSK	08	RR	R1,R2
Set System Mask (p)	SSM	80	SI	D1(B1)
Shift Left Double (c)	SLDA	8F	RS	R1,D2(B2)
Shift Left Double Logical	SLDL	8D	RS	R1,D2(B2)
Shift Left Single (c)	SLA	8B	RS	R1,D2(B2)
Shift Left Single Logical	SLL	89	RS	R1,D2(B2)
Shift Right Double (c)	SRDA	8E	RS	R1,D2(B2)
Shift Right Double Logical	SRDL	8C	RS	R1,D2(B2)
Shift Right Single (c)	SRA	8A	RS	R1,D2(B2)
Shift Right Single Logical	SRL	88	RS	R1,D2(B2)
Start I/O (c,p)	SIO	9C	SI	D1(B1)
Store	ST	50	RX	R1,D2(X2,B2)
Store Character	STC	42	RX	R1,D2(X2,B2)
Store Halfword	STH	40	RX	R1,D2(X2,B2)
Store Multiple	STM	90	RS	R1,R3,D2(B2)
Store Multiple Control (e,p)	STMC	B0	RS	R1,R3,D2(B2)
Subtract (c)	SR	1B	RR	R1,R2
Subtract (c)	S	5B	RX	R1,D2(X2,B2)
Subtract Decimal (c,d)	SP	FB	SS	D1(L1,B1),D2(L2,B2)
Subtract Halfword (c)	SH	4B	RX	R1,D2(X2,B2)
Subtract Logical (c)	SLR	1F	RR	R1,R2
Subtract Logical (c)	SL	5F	RX	R1,D2(X2,B2)
Supervisor Call	SVC	0A	RR	I
Test and Set (c)	TS	93	SI	D1(B1)
Test Channel (c,p)	TCH	9F	SI	D1(B1)
Test I/O (c,p)	TIO	9D	SI	D1(B1)
Test under Mask (c)	TM	91	SI	D1(B1),I2
Translate	TR	DC	SS	D1(L,B1),D2(B2)
Translate and Test (c)	TRT	DD	SS	D1(L,B1),D2(B2)
Unpack	UNPK	F3	SS	D1(L1,B1),D2(L2,B2)
Write Direct (b,p)	WRD	84	SI	D1(B1),I2
Zero and Add (c,d)	ZAP	F8	SS	D1(L1,B1),D2(L2,B2)

NOTES FOR PANELS 1-3

a. Protection feature
b. Direct control feature
c. Condition code is set
d. Decimal feature
e. Model 67
n. New condition code is loaded
p. Privileged instruction
x. Extended precision floating point feature

MACHINE FORMATS

FLOATING-POINT FEATURE INSTRUCTIONS

NAME	MNEMONIC	OP CODE	FORMAT	OPERANDS
Add Normalized, Extended (c,x)	AXR	36	RR	R1,R2
Add Normalized, Long (c)	ADR	2A	RR	R1,R2
Add Normalized, Long (c)	AD	6A	RX	R1,D2(X2,B2)
Add Normalized, Short (c)	AER	3A	RR	R1,R2
Add Normalized, Short (c)	AE	7A	RX	R1,D2(X2,B2)
Add Unnormalized, Long (c)	AWR	2E	RR	R1,R2
Add Unnormalized, Long (c)	AW	6E	RX	R1,D2(X2,B2)
Add Unnormalized, Short (c)	AUR	3E	RR	R1,R2
Add Unnormalized, Short (c)	AU	7E	RX	R1,D2(X2,B2)
Compare, Long (c)	CDR	29	RR	R1,R2
Compare, Long (c)	CD	69	RX	R1,D2(X2,B2)
Compare, Short (c)	CER	39	RR	R1,R2
Compare, Short (c)	CE	79	RX	R1,D2(X2,B2)
Divide, Long	DDR	2D	RR	R1,R2
Divide, Long	DD	6D	RX	R1,D2(X2,B2)
Divide, Short	DER	3D	RR	R1,R2
Divide, Short	DE	7D	RX	R1,D2(X2,B2)
Halve, Long	HDR	24	RR	R1,R2
Halve, Short	HER	34	RR	R1,R2
Load and Test, Long (c)	LTDR	22	RR	R1,R2
Load and Test, Short (c)	LTER	32	RR	R1,R2
Load Complement, Long (c)	LCDR	23	RR	R1,R2
Load Complement, Short (c)	LCER	33	RR	R1,R2
Load, Long	LDR	28	RR	R1,R2
Load, Long	LD	68	RX	R1,D2(X2,B2)
Load Negative, Long (c)	LNDR	21	RR	R1,R2
Load Negative, Short (c)	LNER	31	RR	R1,R2
Load Positive, Long (c)	LPDR	20	RR	R1,R2
Load Positive, Short (c)	LPER	30	RR	R1,R2
Load Rounded, Extended to Long (x)	LRDR	25	RR	R1,R2
Load Rounded, Long to Short (x)	LRER	35	RR	R1,R2
Load, Short	LER	38	RR	R1,R2
Load, Short	LE	78	RX	R1,D2(X2,B2)
Multiply, Extended (x)	MXR	26	RR	R1,R2
Multiply, Long	MDR	2C	RR	R1,R2
Multiply, Long	MD	6C	RX	R1,D2(X2,B2)
Multiply, Long/Extended (x)	MXDR	27	RR	R1,R2
Multiply, Long/Extended (x)	MXD	67	RX	R1,D2(X2,B2)
Multiply, Short	MER	3C	RR	R1,R2
Multiply, Short	ME	7C	RX	R1,D2(X2,B2)
Store, Long	STD	60	RX	R1,D2(X2,B2)
Store, Short	STE	70	RX	R1,D2(X2,B2)
Subtract Normalized, Extended (c,x)	SXR	37	RR	R1,R2
Subtract Normalized, Long (c)	SDR	2B	RR	R1,R2
Subtract Normalized, Long (c)	SD	6B	RX	R1,D2(X2,B2)
Subtract Normalized, Short (c)	SER	3B	RR	R1,R2
Subtract Normalized, Short (c)	SE	7B	RX	R1,D2(X2,B2)
Subtract Unnormalized, Long (c)	SWR	2F	RR	R1,R2
Subtract Unnormalized, Long (c)	SW	6F	RX	R1,D2(X2,B2)
Subtract Unnormalized, Short (c)	SUR	3F	RR	R1,R2
Subtract Unnormalized, Short (c)	SU	7F	RX	R1,D2(X2,B2)

SUMMARY OF CONSTANTS

TYPE	IMPLIED LENGTH, BYTES	ALIGNMENT	FORMAT	TRUNCATION/PADDING
C	–	byte	characters	right
X	–	byte	hexadecimal digits	left
B	–	byte	binary digits	left
F	4	word	fixed-point binary	left
H	2	halfword	fixed-point binary	left
E	4	word	short floating-point	right
D	8	doubleword	long floating-point	right
L	16	doubleword	extended floating-point	right
P	–	byte	packed decimal	left
Z	–	byte	zoned decimal	left
A	4	word	value of address	left
Y	2	halfword	value of address	left
S	2	halfword	address in base-displacement form	–
V	4	word	externally defined address value	left
Q*	4	word	symbol naming a DXD or DSECT	left

*OS only

图 1-2 IBM 360 架构指令集图卡

1.1.2 CISC 与 RISC

指令集架构主要分为复杂指令集（Complex Instruction Set Computer，CISC）和精简指令集（Reduced Instruction Set Computer，RISC），两者的主要区别如下。

- CISC 不仅包含了处理器常用的指令，还包含了许多不常用的特殊指令。其指令数目比较多，所以称为复杂指令集。
- RISC 只包含处理器常用的指令，而对于不常用的操作，则通过执行多条常用指令的方式来达到同样的效果。由于其指令数目比较精简，所以称为精简指令集。

在 CPU 诞生的早期，CISC 曾经是主流，因为其可以使用较少的指令完成更多的操作。但是随着指令集的发展，越来越多的特殊指令被添加到 CISC 指令集中，CISC 的诸多缺点开始显现出来。譬如：

- 典型程序的运算过程中所使用到的 80%指令，只占所有指令类型的 20%，也就是说，

CISC 指令集定义的指令，只有 20%被经常使用到，而有 80%则很少被用到。

- 那些很少被用到的特殊指令尤其让 CPU 设计变得极为复杂，大大增加了硬件设计的时间成本与面积开销。

基于以上原因，自从 RISC 诞生之后，基本上所有现代指令集架构都选择使用 RISC 架构。

1.1.3 32 位与 64 位架构

除了 CISC 与 RISC 之分，处理器指令集架构的位数也是一个重要的概念。通俗来讲，处理器架构的位数是指通用寄存器的宽度，其决定了寻址范围的大小、数据运算能力的强弱。譬如 32 位架构的处理器，其通用寄存器的宽度为 32 位，能够寻址的范围为 2^{32}，即 4GB 的寻址空间，运算指令可以操作的操作数为 32 位。

注意：处理器指令集架构的宽度和指令的编码长度无任何关系。并不是说 64 位架构的指令长度为 64 位（这是一个常见的误区）。从理论上来讲，指令本身的编码长度越短越好，因为可以节省代码的存储空间。因此即便在 64 位的架构中，也大量存在 16 位编码的指令，且基本上很少出现过 64 位长的指令编码。

综上所述，在不考虑任何实际成本和实现技术的前提下，理论上来讲：

- 通用寄存器的宽度，即指令集架构的位数越多越好，因为这样可以带来更大的寻址范围和更强的运算能力。
- 指令编码的长度越短越好，因为这样可以更加节省代码的存储空间。

常见的架构位数分为 8 位、16 位、32 位和 64 位。

- 早期的单片机以 8 位和 16 位为主，譬如知名的 8051 单片机是使用广泛的 8 位架构。
- 目前主流的嵌入式微处理器均在向 32 位架构转移。对此内容感兴趣的读者可以在互联网上搜索作者曾在媒体上发表的文章《进入 32 位时代，谁能成为下一个 8051》。
- 目前主流的移动手持、个人计算机和服务器领域，均使用 64 位架构。

有关嵌入式、移动手持、个人计算机和服务器领域的详情，请参见第 1.1.5 节关于 CPU 领域之分的介绍。

1.1.4 ISA 众生相

第 1.1.1 节中提到，经过几十年的发展，全世界范围内至今已经相继诞生或消亡过了几十种不同的指令集架构。下面将针对几款比较知名的指令集架构加以论述。

注意：下列章节中列举的信息来自于本书成书时的公开信息，非官方正式信息，请读者以最新官方信息为准。

1. x86

x86 是由 Intel 公司推出的一种复杂指令集（CISC），于 1978 年推出的 Intel 8086 处理器

中首度出现，如图 1-3 所示。8086 在 3 年后为 IBM 所选用，之后 Intel 与微软公司结成了所谓的 Windows-Intel（Wintel）商业联盟，垄断了个人计算机（Personal Computer，PC）软硬件平台至今几十年而获得了丰厚的利润。x86 架构也因此几乎成为了个人计算机的标准处理器架构，而 Intel 的广告标志更是深入人心，如图 1-4 所示。

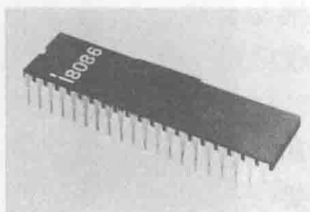


图 1-3　Intel 8086 处理器

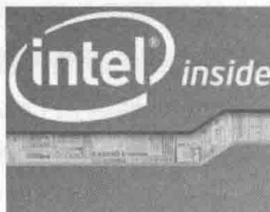


图 1-4　Intel Inside 广告标语

除 Intel 之外最成功的制造商之一为 AMD。Intel 与 AMD 公司是现今主要的 x86 处理器芯片提供商。其他若干几个公司也曾经制造过 x86 架构的处理器，包括 Cyrix（为 VIA 所收购）、NEC、IBM、IDT 以及 Transmeta。

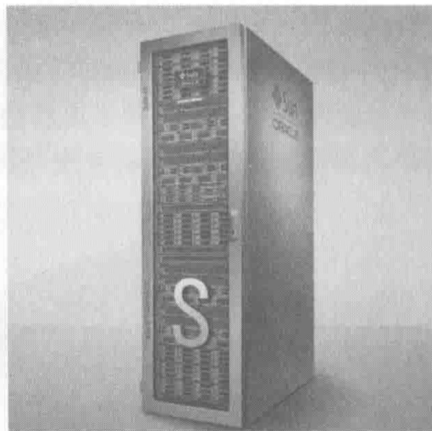
x86 架构由 Intel 与 AMD 共同经过数代的发展，相继从最初的 16 位架构发展到如今的 64 位架构。在 x86 架构刚诞生的时代，CISC 还是业界主流，因此，x86 架构是具有代表性的可变指令长度的 CISC 指令架构。虽然之后 RISC 已经取代 CISC 成为现代指令集架构的主流，但是，由于 Intel 公司的巨大成功以及为了维护软件的向后兼容性，x86 作为一种 CISC 架构被一直保留下来。事实上，Intel 公司通过内部“微码化”的方法克服掉了 CISC 架构的部分缺点，加上 Intel 高超的 CPU 设计水平与工艺制造水平，使得 x86 处理器一直保持着旺盛的战斗力，不断刷新个人计算机处理器芯片性能的极限。所谓“微码化”是指将复杂的 CISC 指令先用硬件解码器翻译成对应的内部简单指令（微码）序列，然后送给处理器流水线执行的方法，使得 x86 的处理器核也变成了一种 RISC 的形式，从而能够借鉴 RISC 架构的优点。不过，额外的硬件解码器同样也会带来额外的复杂度与面积开销，这是 x86 架构作为一种 CISC 架构不得不付出的代价。

x86 架构不仅在个人计算机领域取得了统治性的地位，还在服务器市场取得了巨大成功。相比 x86 架构，IBM 的 Power 架构和 Sun 的 SPARC 架构都曾有着很明显的性能优势，也曾占据着相当可观的服务器市场。但是 Intel 采用仅提供处理器芯片而不直接生产服务器的策略，利用广大的第三方服务器生产商，结合 Wintel 的强大软硬件联盟，成功地将从处理器芯片到服务器系统一手包办的 IBM 与 Sun 公司击败。至今 x86 架构占据了超过 90%的服务器市场。

2. SPARC

1985 年，Sun 公司设计出 SPARC 指令集架构，全称为“可扩充处理器架构（Scalable Processor ARChitecture，SPARC）”，是一种非常有代表性的高性能 RISC 架构。之后，Sun 公司和 TI 公司合作开发了基于该架构的处理器芯片。SPARC 处理器为 Sun 公司赢得了当时

高端处理器市场的领先地位。1995 年，Sun 公司推出了 UltraSPARC 处理器，开始进入 64 位架构。SPARC 架构设计的出发点是服务于工作站，它被应用在 Sun、富士通等制造的大型服务器上，如图 1-5 所示。1989 年 SPARC 还作为独立的公司而成立，其目的是向外界推广 SPARC，以及为该架构进行兼容性测试。Oracle 收购 Sun 公司之后，SPARC 架构归 Oracle 所有。


图 1-5 基于 SPARC 架构的服务器

由于 SPARC 架构是面向服务器领域而设计的，其最大的特点是拥有一个大型的寄存器窗口，SPARC 架构的处理器需要实现从 72 到 640 个之多的通用寄存器，每个寄存器宽度为 64 位，组成一系列的寄存器组，称为寄存器窗口。这种寄存器窗口的架构由于可以切换不同的寄存器组快速地响应函数调用与返回，因此能够带来非常高的性能，但是这种架构由于功耗面积代价太大而并不适用于 PC 与嵌入式领域处理器。

前面提到了 Sun 公司在服务器领域与 Intel 竞争逐渐落败，因此，SPARC 架构在服务器领域的份额也逐步地缩减。而 SPARC 架构又不适用于 PC 与嵌入式领域，使得其局面十分尴尬。

SPARC 架构应用的另外一个比较知名的领域是航天领域。由于美国的航天星载系统中普遍使用的 Power 架构（有关 Power 架构参见本书后面关于 Power 的介绍），欧洲太空局为了独立发展自己的航天能力而选择了开发基于 SPARC 架构的 LEON 处理器，并对其进行了抗辐射加固设计，使之能够应用于航天环境中。

值得强调的是，欧洲太空局选择在航天领域使用 SPARC 架构并不代表 SPARC 架构特别适用于航天领域，而是因为 SPARC 在当时是一种相对开放的架构。SPARC 架构也更罔谈垄断或占据航天领域的优势地位，因为从本质上来讲，航天领域处理器对于指令集架构本身并无特殊要求，其需求的主要特性是提供工艺上的加固单元和硬件系统的容错性处理（为了防止外太空强辐射造成电路失常）。因此，很多的航天处理器也采用了其他的处理器架构，譬如目前新开发的很多航天处理器也在使用新的 ARM 或者 RISC-V 架构（有关 RISC-V 架构参见第 1.5 节）。

2017 年 9 月，Oracle 公司宣布正式放弃硬件业务，自然也包括了收购自 Sun 的 SPARC 处理器，至此，SPARC 处理器可以说正式退出了历史舞台。此消息一经发出，引起了很多业内人士的惋惜，感兴趣的读者请在网络上自行搜索文章《再见 SPARC 处理器，再见 Sun》。

3. MIPS

MIPS（Microprocessor without Interlocked Piped Stages Architecture）亦为 Millions of Instructions Per Second 的相关语，是一种简洁、优化的 RISC 架构。MIPS 可以说是身出名门，

由斯坦福大学的 Hennessy 教授（计算机体系结构领域泰斗之一）领导的研究小组研制开发。

由于 MIPS 是经典的 RISC 架构，因此是如今除了 ARM 之外被人耳熟能详的 RISC 架构。最早的 MIPS 架构是 32 位，最新的版本已有 64 位。

自从 1981 年由 MIPS 科技公司开发并授权后，MIPS 架构曾经作为最受欢迎 RISC 架构被广泛应用在许多电子产品、网络设备、个人娱乐装置与商业装置上。它曾经在嵌入式设备与消费领域里占据很大的份额，如 SONY、Nintendo 的游戏机、Cisco 的路由器和 SGI 超级计算机都有 MIPS 的身影。

但是由于一些商业运作的原因，MIPS 被同属 RISC 阵营的 ARM 后来居上。2013 年 MIPS 被英国公司 Imagination Technologies 收购，可惜的是，MIPS 被 Imagination 收购后，非但没有发展，反而日渐衰落。2017 年 Imagination 自身出现危机而整体寻求出售，MIPS 再次面临被出售的命运。

4. Power

Power 架构是 IBM 开发的一种 RISC 架构指令集。1980 年 IBM 推出了全球第一台基于 RISC 架构的原型机，证明 RISC 相比 CISC 在高性能领域优势明显。1994 年 IBM 基于此推出 PowerPC604 处理器，其强大的性能在当时处于全球领先地位。

基于 Power 架构的 IBM Power 服务器系统在可靠性、可用性和可维护性等方面表现出色，使得 IBM 从芯片到系统所设计的整机方案有着独有的优势。Power 架构的处理器在超算、银行金融、大型企业的高端服务器等多个方面应用十分成功。IBM 至今仍在不断开发新的 Power 架构处理器：

- 2013 年，IBM 宣布了新一代服务器处理器 Power8。Power8 的核心数量达 12 个，而且每个核心都支持 8 线程，总线程多达 96 个。它采用了 8 派发、10 发射、16 级流水线的设计，各项规格均强大得令人惊叹。
- 2016 年 IBM 公司公布了其 Power9 处理器，IBM 于 2017 年推出 Power9 拥有 24 个计算核心，是 Power8 芯片的两倍。
- IBM 计划在 2020 年推出 Power10，2023 年推出 Power11 处理器。

5. Alpha

Alpha 也称为 Alpha AXP，是一种 64 位的 RISC 指令集架构，由 DEC 公司设计开发，被用于 DEC 自己的工作站和服务器中。

Alpha 是一款优秀的处理器，它不仅是最早跨过 GHz 的企业级处理器，而且还是最早计划采用双核，甚至是多核架构的处理器。然而，Alpha 芯片和采用此芯片的服务器并没有得到整个市场的认同，只有少数人选择了 Alpha 服务器。据称其价格高昂、安装复杂，部署实施远远超过一般企业 IT 管理人员所能承受的难度。2001 年，康柏收购 DEC 之后，逐步将其全部 64 位服务器系列产品转移到 Intel 的安腾处理器架构之上。2004 年，惠普收购康柏，

从此 Alpha 架构逐渐淡出了人们的视野。

6. ARM

由于 ARM 架构过于声名显赫，后续会有专门的小节重点论述 ARM，在此不单独论述。

7. ARC

ARC 架构处理器是 Synopsys 公司推出的 32 位 RISC 结构微处理器系列 IP。ARC 处理器的 IP 产品线覆盖了从低端到高端各个领域的嵌入式处理器，如图 1-6 所示。



图 1-6 ARC 处理器 IP 系列

ARC 架构处理器以极高的能效比见长，出色的硬件微架构使得 ARC 处理器的各项指标均令人印象深刻。ARC 处理器 IP 以追求功耗效率比（DMIPS/mW）和面积效率比（DMIPS/mm^2）最优化为目标，以满足嵌入式市场对微处理器产品日益提高的效能要求。

ARC 处理器的另外一个最大的特点是其高度可配置性，可通过增加或删除功能模块，满足不同的应用需求，通过配置不同属性实现快速系统集成，做到“量体裁衣”。

ARC 是除了 ARM 之外的全球第二大嵌入式处理器 IP 供应商，全球已有超过 170 家客户使用 Synopsys ARC 处理器，这些客户每年总共产出高达 15 亿块基于 ARC 的芯片。

8. Andes

Andes 架构处理器是中国台湾省的晶心（Andes）公司推出的一系列 32 位 RISC 架构处理器 IP。据 2016 年的统计数字，采用 Andes 指令集架构的系统芯片出货量超过 4.3 亿颗，总累计出货量超过 19 亿颗。2017 年，Andes 发布最新一代的 AndeStar™处理器架构，成为商用主流 CPU IP 公司中第一家纳入开放 RISC-V 指令集架构的公司。

9. C-Sky

C-SKY 架构处理器是由杭州中天微系统有限公司开发的一系列 32 位高性能低功耗嵌入式处理器 IP。杭州中天是国内 CPU IP 公司的翘楚，C-SKY 系列嵌入式 CPU 核，具有低功耗、高性能、高代码密度、易使用等特点。

1.1.5 CPU 的领域之分

本节将对 CPU 的不同应用领域加以探讨。

在传统的计算机体系结构分类中，处理器应用分为 3 个领域——服务器领域、PC 领域和嵌入式领域。

- 服务器领域在早期还存在着多种不同的架构呈群雄分立之势，不过，由于 Intel 公司商业策略上的成功，目前 Intel 的 x86 处理器芯片几乎成为了这个领域的霸主。

- PC 领域本身是由 Windows/Intel 软硬件组合发展而壮大，因此，x86 架构是目前 PC 领域的垄断者。
- 传统的嵌入式领域所指范畴非常广泛，是处理器除了服务器和 PC 领域之外的主要应用领域。所谓“嵌入式”是指在很多芯片中，其所包含的处理器就像嵌入在里面不为人知一样。

近年来随着各种新技术新领域的进一步发展，嵌入式领域本身也被发展成了几个不同的子领域而产生了分化。

- 首先是随着智能手机（Mobile Smart Phone）和手持设备（Mobile Device）的发展，移动（Mobile）领域逐渐发展成了规模匹敌甚至超过 PC 领域的一个独立领域，其主要由 ARM 的 Cortex-A 系列处理器架构所垄断。由于 Mobile 领域的处理器需要加载 Linux 操作系统，同时涉及复杂的软件生态，因此，其具有和 PC 领域一样对软件生态的严重依赖。目前既然 ARM Cortex-A 系列已经取得了绝对的统治地位，其他的处理器架构很难再进入该领域。
- 其次是实时（Real Time）嵌入式领域。该领域相对而言没有那么严重的软件依赖性，因此没有形成绝对的垄断，但是由于 ARM 处理器 IP 商业推广的成功，目前仍然以 ARM 的处理器架构占大多数市场份额，其他处理器架构譬如 Synopsys ARC 等也有不错的市场成绩。
- 最后是深嵌入式领域。该领域更像前面所指的传统嵌入式领域。该领域的需求量非常之大，但往往注重低功耗、低成本和高能效比，无须加载像 Linux 这样的大型应用操作系统，软件大多是需要定制的裸机程序或者简单的实时操作系统，因此对软件生态的依赖性相对比较低。在该领域很难形成绝对的垄断，但是由于 ARM 处理器 IP 商业推广的成功，目前仍然以 ARM 的 Cortex-M 处理器占据大多数市场份额，其他的架构譬如 Synopsys ARC 和 Andes 等也有非常不错的表现。

综上所述，由于移动（Mobile）领域崛起成为一个独立的分类领域，现在通常所指的嵌入式领域往往是指深嵌入式领域或者实时嵌入式领域。

表 1-2 是对目前 CPU 典型应用领域及其主流的架构进行的总结。

表 1-2 处理器的应用领域及主流架构

领　域	主流架构
服务器（Server）领域	Intel 公司 x86 架构的高性能 CPU 占垄断地位
桌面个人计算机（PC）领域	Intel 或者 AMD 公司 x86 架构的 CPU 占垄断地位
嵌入式移动手持设备（Mobile）领域	ARM Cortex-A 架构占垄断地位
嵌入式实时设备（Real Time）领域	ARM 架构占最大份额，其他 RISC 架构的嵌入式 CPU 也有不错的表现
深嵌入式（Deep Embedded）领域	ARM 架构占最大份额，其他 RISC 架构的嵌入式 CPU 也有不错的表现

1.2 ISA 请扛起这口锅——为什么国产 CPU 尚未足够成功

众所周知，芯片是我国信息产业发展的核心领域，而 CPU 则代表了芯片中的核心技术。在此方面，我国与发达国家相比有着明显的差距。虽然经过多年的努力，技术差距已经有了显著的缩小，但是在民用商业领域内，仍然没有看到太多国产 CPU 的身影。是什么原因造成国产商业 CPU 尚未足够成功这一现状呢？接下来，我们便细数一下国内自主开发 CPU 的公司与现状，以及它们选择的指令集流派。通过逐一分析其过去与现状，相信能够让读者得到答案。

1.2.1 MIPS 系——龙芯和君正

1. 龙芯

龙芯 CPU 由中国科学院计算技术所龙芯课题组研制，由中国科学院计算技术所授权的北京神州龙芯集成电路设计公司研发。以下是龙芯 CPU 芯片的相关简介。

- 龙芯 1 号的频率为 266MHz，最早在 2002 年开始使用，如图 1-7 所示。
- 龙芯 2 号的频率最高为 1GHz。
- 龙芯 3A 系列是国产商用 4 核处理器。最新龙芯 3A3000 基于中芯 28nm FDSOI 工艺，设计为 4 核 64 位，主频为 1.5GHz，功耗仅为 30W，非常适合笔记本平台。



图 1-7 龙芯处理器芯片

- 龙芯 3B 系列是国产商用 8 核处理器，主频超过 1GHz，支持向量运算加速，峰值计算能力达到每秒 1.28×10^3 亿次浮点运算，具有很高的性能功耗比。龙芯 3B 系列主要用于高性能计算机、高性能服务器、数字信号处理等领域。

2. 君正

国内的 MIPS 系还有另外一家公司——北京君正。君正和龙芯同属于 MIPS 阵营，与龙芯着力于桌面 PC 处理器不同，北京君正是国内较早专注于可穿戴、物联网领域的本土 IC 设计公司之一。由于嵌入式芯片的软件一般按需求定制。这导致在智能可穿戴市场，相当一部分可穿戴产品和应用软件具有专用性，软件生态链相对较短，加上应用需求的多样化，因此不能用一套通用方案来满足所有人的要求，所以在这个领域没有某个厂商可以实现垄断。因此，在智能穿戴市场不容易出现 PC 和移动手机市场那样被 x86 与 ARM 架构垄断的情况。

智能穿戴芯片和物联网芯片对性能要求不高，大部分应用场景更关注低功耗、廉价和尺

寸等因素，君正的产品完全满足性能要求，x86 处理器不可能应用于该领域，ARM 阵营 IC 设计公司受制于相对较高的授权费，在芯片产量较小的情况下，并不具备价格上的竞争力。君正拥有十多年的芯片设计经验和技术积累，其最大的特点就是具有较高的性能功耗比。国内第一批上市的智能手表包括果壳的第一代智能手表、土曼一代、土曼二代智能手表等都采用了君正的方案。

1.2.2 x86 系——北大众志、兆芯和海光

1. 北大众志

北京北大众志微系统科技有限责任公司成立于 2002 年 11 月，是国家集成电路设计行业的重要骨干企业。2005 年，AMD 与中国政府达成了协议，科技部指定北大微电子中心接收 AMD Geode-2 处理器的技术授权，AMD 的处理器无疑是 x86 架构，中国因此获得了 x86 技术。不过 Geode 处理器属于 AMD 嵌入式处理器，因此 AMD 授权给北大的 x86 技术属于嵌入式架构。

2. 兆芯

另外一家使用 x86 架构的国内企业——兆芯，也许被更多的人所熟知。众所周知，核心的 x86 架构是 Intel 和 AMD 公司的核心技术，美国政府也会严格控制其技术的授权。不过，除了 Intel 和 AMD，另外一家中国台湾公司威盛（VIA）也曾经拥有 x86 架构授权。如图 1-8 所示，兆芯自主研发的 ZX-C 处理器于 2015 年 4 月量产，28nm 工艺，4 核处理器，主频可达 2.0GHz，并且支持国密算法加密。2017 年兆芯宣布其最新一代 ZX-D 系列 4 核和 8 核通用处理器已经成功流片，并透露将在 2018 年推出 16nm 的 ZX-E 8 核 CPU。



图 1-8　兆芯处理器芯片

3. 海光

除了上海兆芯，还有一家诞生不久的新锐公司——天津海光。2016 年，AMD 宣布与中国天津海光投资公司达成了协议，将 x86 技术授权给海光公司，获得授权费，并且双方还会成立合资公司，授权其生产服务器处理器。据称，为了打开中国高性能服务器市场，AMD 这次授权给中国公司的 x86 很可能是最尖端的 x86 技术。对于海光的表现，也值得我们拭目以待。

1.2.3 Power 系——中晟宏芯

蓝色巨人 IBM 的 Power 架构一直是高性能的代言。IBM 于 2013 年联合 NVIDIA 等公司

成立 OpenPower 开放联盟，其他公司也可以获得 Power 架构授权。此后还推动成立了中国 POWER 技术产业生态联盟，与多家中国公司签署了授权协议，中晟宏芯就是其中的一家。中晟宏芯成立于 2013 年，相信宏芯能用若干年的时间实现技术的消化吸收和推陈出新。

1.2.4 Alpha 系——申威

申威处理器或申威 CPU，简称“SW 处理器”。

申威对自主的 Alpha 架构在不断深化升级，在双核 Alpha 基础上拓展了多核架构和 SIMD 等特色扩展指令集，主要面向高性能计算、服务器领域，在 2016 年国际超算大会评比中，基于申威 26010 处理器的“神威太湖之光”超级计算机系统（如图 1-9 所示）首次亮相并夺冠，其峰值性能达每秒 12.5×10^8 亿次浮点运算，成为世界首台运行速度超 10^9 亿次的超级计算机。



图 1-9　基于申威处理器的神威太湖之光超级计算机

1.2.5 ARM 系——飞腾、华为海思、展讯和华芯通

为了更好地理解本节的内容，有必要先对 ARM 的授权模式进行介绍。简而言之，ARM 公司的主要授权模式可以分为两种。

- 授权“ARM 处理器 IP”给其他的芯片生产商（合作伙伴），后者直接使用 ARM 处理器 IP 设计 SoC 芯片。
- 授权“ARM 架构”给其他的芯片生产商（合作伙伴），后者基于 ARM 架构自研其处理器核，然后使用自研处理器核设计 SoC 芯片。

请参见第 1.4.1 节了解更多信息。

注意：购买 ARM 的处理器 IP 进行 SoC 芯片开发的国内公司众多，但是由于其直接使用 ARM 公司的处理器 IP，并不属于真正的自主“处理器核”开发，所以不在本书的讨论之列。而授权“ARM 架构”进行处理器核的自研，目前国内只有飞腾、海思和展讯等具备这样的实力，下面分别予以介绍。

1. 飞腾

飞腾公司是中国国防科技大学高性能处理器研究团队建立的企业，国防科大多年来在 CPU 领域的耕耘积累了雄厚的技术实力。2016 年天津飞腾公布了最新产品 FT2000，它最早亮相于 2015 年的 HotChips 大会，代号“火星”，定位于高性能服务器、行业业务主机等。FT2000 采用 ARMv8 指令集，但是使用自研内核，不同于市面上 ARMv8 的 Cortex-A53\A57\A72（直

接购买于 ARM 公司的内核）。

FT2000 之所以引人注目还因为它在性能方面，包括高达 64 个 FTC661 处理器核，其公布的 Spec 2006 测试中，成绩为整数 672、浮点 585，足以和 Xeon E5－2699v3 相媲美。这也是国产服务器芯片第一次在性能上追平 Intel，存储器控制芯片总聚合带宽为 204.8GB/s，超过目前的 E5V3 和 E7V3，接近 IBM POWER8（230GB/s）。跑分与 Intel 的 Xeon E5－2699v3 相媲美意味着飞腾 2000 对于很多商业应用来说已经完全够用了，只要软件生态跟得上，完全可以在商业市场上取代 Intel 的某些产品。

2．华为海思

华为海思目前是我国技术最强大的芯片开发商之一。华为的麒麟芯片在性能上与高通、三星这些领先的芯片企业处于一个水平。同时华为目前也是国内四大服务器提供商之一，华为、联想、浪潮等国产服务器企业占有中国服务器市场的份额已经超过 65%。华为在几年前便已经购买了 ARM 指令集架构授权，开始研发自有的处理器核，主攻服务器市场。

在“十二五”科技创新成就展上，华为展出了其第一台 ARM 平台服务器“泰山”，配备自主研发 ARM 架构 64 位处理器“Hi1612”，采用台积电 16nm 工艺，拥有多达 16 个核心，兼容 ARMv8-A 指令集。凭借华为强大的研发实力与市场运作能力，相信一定会有不俗的表现。

3．展讯

除华为之外，展讯是另一家国内手机芯片的翘楚。2016 年展讯的芯片出货达到 6.7 亿套，2017 年 6 月宣布成功研发其自主的 ARM 架构处理器，展讯宣称在 SC9850 4 核（Cortex-A7）芯片同样大的面积上实现了 6 核的设计，功耗和性能都可以按照自己的需求调配，标志着展讯成为了除苹果、三星两家智能手机厂商之外（三星和苹果的自主芯片主要都是自用），继高通之后，第二家拥有自主 ARM CPU 关键技术的手机芯片厂商。

4．华芯通

2016 年，高通与中国贵州政府合资在华成立了一家芯片公司——华芯通半导体，旨在专门为中国市场设计与开发服务器专用芯片的公司。华芯通已获 ARM v8-A 架构授权，并表示中国成为全球第二大数据中心市场，该授权将帮助华芯通半导体在快速扩张的中国服务器市场推出先进服务器芯片组技术，帮助中国企业在本土市场提供基于 ARM 的服务器技术，从而推动高效服务器解决方案的大规模部署。

1.2.6 背锅侠 ISA

从上述几个章节中，我们已经了解了国内 CPU 设计的英雄榜。但是如前文所述，目前在民用商业领域内，仍然没有看到太多国产 CPU 的身影。可以说，国产处理器在民用商业领域至今尚未足够成功的主要原因在于 ISA，这口锅 ISA 必背无疑。

在第 1.1.1 节中已经论述了指令集架构（ISA）对于 CPU 的重要性，那么对于一款 CPU 而言，绝对的硬件技术水平不是最重要的。

目前商业主流的指令集架构在不同的领域已经各自出现了明显的霸主格局。

- x86 架构统治着桌面 PC 与服务器领域。
- ARM 架构统治着移动手持领域，同时对桌面 PC 和服务器领域全面进军。
- ARM 在嵌入式领域占据绝对优势。

因此作者之前一直认为，只有依附于 x86 与 ARM 阵营的商业公司，才能够真正地实现全面的商用化。相信这也是近几年来国内 CPU 设计的英雄榜上涌现出来的大多为 x86 或者 ARM 系的原因。

但是，国产自主对我国的国计民生又至关重要，追求国产自主安全可控是我国在战略上必须坚持的方向。从这个角度上来看，选择 x86 或者 ARM 架构终究也有其局限性，分别论述如下。

1. x86 架构

- 由于 Intel 与 AMD 本身是芯片公司而不是知识产权（IP）公司，因此 x86 架构是其生命线，假设其他得到授权的芯片公司使用 x86 架构生产的芯片对 Intel 和 AMD 造成了实质威胁时，Intel 与 AMD 完全可以拿起专利的大棒停止授权。
- x86 架构的授权费用极为高昂，远非普通公司或者组织能够染指。

2. ARM 架构

- ARM 架构的局面会乐观很多，因为 ARM 架构虽然也是属于 ARM 公司且受专利保护的架构，但是 ARM 公司的商业模式是以开放共赢为基本原则。ARM 公司是 ARM 生态的主导者和核心规则的制定者，通过基础架构授权、IP 核授权等方式获得经济收益。而生态系统中大量的上下游软硬件企业则遵循 ARM 统一制定的标准规范，对接众多客户需求而实现经济利益的获取。
- 国内基于 ARM 生态的 CPU 产业已有较好基础，华为海思、展讯、联芯和飞腾等众多企业均已累积多年的 ARM 芯片研发经验，在移动终端领域我国芯片设计技术已与国际主流水平同步，国外的巨头高通、三星和谷歌等也属于 ARM 生态系统阵营的成员。因此，从全球范围来看，国内外的芯片公司能够在开放共赢的生态下进行公平的竞争。基于上述原因，国内 CPU 英雄榜上使用 ARM 架构的 CPU 公司，其成就更加令人可期。
- 尽管如此，ARM 架构毕竟属于 ARM 公司，一方面需要为 ARM 公司支付极其高昂的授权费（一次数千万美金），另一方面被软银收购后 ARM 现在属于一家日本公司。因此，从绝对的自主可控的角度来看，受制于人那是在所难免的。

所谓“成也萧何，败也萧何”，读到此处，读者可能要问，难道就没有一种 ISA 具备如

下几个特点吗？

（1）它开源共享，不属于某一家商业公司私有，因此也就不会有受制于人与自主可控的隐忧，更加不需要向商业公司支付高昂的授权费。

（2）它以开放共赢为基本原则，有一个统一的非营利组织作为主导者和核心规则的制定者，任何公司和个人都可以永久免费地使用其架构。

- 生态系统中大量的上下游软硬件企业应遵循该组织统一制定的标准规范，对接众多客户需求而实现经济利益的获取。
- 同样从全球范围来看，国内国外的芯片公司能够在此开放共赢的生态下进行公平的竞争。

相信很多人都与作者一样，在很长的一段时间内，非常期待有这样一种 ISA 的出现，业界甚至出现过希望由国家主导指定一种国家标准 ISA，从而统一国内 CPU 各 ISA 派系的声音。然而，国家标准 ISA 这种被局限在一国范围内的技术在当今全球化的趋势下，必然是格格不入且不可能成功的。于是所有人都认为不可能出现这样一种 ISA 了，作者作为一名 CPU 设计的老兵，也不得不用一首诗来表达一下彼时的心情：“死去元知万事空，但悲不见九州同。王师北定中原日，家祭无忘告乃翁”。

然而在 2016 年，有一位叫作 RISC-V 的新生突然自带光环登场。它完全符合上述提到的两个条件，属于全人类的免费开放架构，无任何专利的桎梏，众多国际知名大公司均加入其中，将以开放共赢的生态下进行公平的竞争。作者隐隐感到，如果这个 ISA 真能够发展起来，这似乎可能是国产 CPU 崛起的真正机会。刚才我们提到曾有人建议制定一种国家标准的指令集架构，而当 RISC-V 诞生不久，我们的邻国印度迅速地采用了 RISC-V 作为其国家标准的指令集，推荐其国内的大学和研究机构均采用 RISC-V 架构，并且已经制定规划且投入专项资金用于开发几个不同系列的 RISC-V 处理器。

有道是“山重水复疑无路，柳暗花明又一村”，有关新生的 RISC-V 架构，我们将在第 1.5 节中详细介绍。

1.3 人生已是如此艰难，你又何必拆穿——CPU 从业者的无奈

对于每一个行业的普通从业者而言，都希望所在行业能够蓬勃发展、欣欣向荣，能够有大量的商业公司参与并产生大量工作岗位的需求。倘使所在的行业或是日暮西山，或是走向寡头化成为一潭死水，自然也就无法诞生大量的工作需求，那普通的从业者们可能就只有“寻寻觅觅，冷冷清清，凄凄惨惨戚戚”，或者“门前冷落鞍马稀，老大嫁作商人妇”了。

处理器设计便是一个典型的例子。虽然处理器设计是一门开放的学科，其所需的技术均已成熟，很多的工程师与从业人员都已经掌握，也具备开发的处理器的能力。但是：

- 由于处理器架构长期以来主要由以 Intel（x86 架构）与 ARM（ARM 架构）为代表的商业巨头公司所掌控，及其软件生态环境衍生出的寡头排他效应，成为了普通公司与个人无法逾越的天堑。
- 由于寡头的排他效应，众多的处理器体系结构走向消亡，国产的商用 CPU 也无法足够成功，从而造成了 CPU 设计这项工作变成了极少数商业公司的“堂前燕”，普通平民“只可远观，而不可亵玩焉”，国内长期没有形成有足够影响力的相关产业与商业公司。

综上，作者作为曾经在国际一流公司任职的 CPU 高级设计工程师，竟一度在换工作时面临择业无门的窘境，更扼腕叹息众多同仁被迫转行的情形。正可谓“曲高者和寡，大音者稀声”，CPU 设计从业者颇无奈也。读至此，被迫转行的同仁们可能已经老泪纵横：“人生已是如此的艰难，你又何必拆穿啊”。

好消息是最近几年来国内 CPU 产业的情形终于发生了的改观，由于中国的巨大市场与产业支持，国内涌现出了如上节中我们提到的兆芯、飞腾、华为、展讯、海光和华芯通等从事 CPU 设计的公司，且随着本书介绍的 RISC-V 架构之诞生，都将催生更多的市场需求。

1.4 无敌是多么寂寞——ARM 统治着的世界

ARM（Advanced RISC Machines）是一家诞生于英国的处理器设计与软件公司，总部位于英国的剑桥，其主要业务是设计 ARM 架构的处理器，同时提供与 ARM 处理器相关的配套软件，各种 SoC 系统 IP、物理 IP、GPU、视频和显示等产品。

虽然在普通人眼中，ARM 公司的知名度远没有 Intel 公司的辨识度高，甚至不如华为、高通、苹果、联发科和三星等这些厂商那般耳熟能详。但是，ARM 架构处理器却以“润物细无声”的方式渗透到我们生活中的每个角落。从我们每天日常使用的电视、手机、平板电脑以及手环、手表等电子产品，到不起眼的遥控器、智能灯和充电器等我们想到和想不到的方方面面，均有 ARM 架构处理器的身影。在白色家电、工业控制与汽车电子领域，ARM 架构处理器更是无处不在；乃至我们熟知桌面 PC、服务器和超级计算机领域，ARM 架构也在朝其渗透。可以说，ARM 架构处理器统治着这些领域，支撑着我们这个世界的运行。

1.4.1 独乐乐与众乐乐——ARM 公司的盈利模式

ARM 公司虽然设计开发基于 ARM 架构的处理器核，但是商业模式并不是直接生产处理器芯片，而是作为知识产权（Intellectual Property，IP）供应商，转让授权许可给其合作伙伴。目前，全世界有几十家大的半导体公司都使用 ARM 公司的授权，从 ARM 公司购买其

设计的 ARM 处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 处理器芯片进入市场。

至此，我们提到了若干名词“ARM 架构”“ARM 架构处理器”或“ARM 处理器”“ARM 处理器芯片”“芯片”。为了能够阐述清楚其彼此的关系，并理解 ARM 公司的商业模式，下面通过一个形象的比喻加以阐述。

假设将“生产芯片”比喻为“制造一辆汽车”，我们知道在市场上有几十家品牌汽车生产商，譬如“大众”“丰田”“本田”等。那么芯片领域也有众多的芯片生产商，譬如“高通”“联发科”“三星”“德州仪器”等。有的芯片是以处理器的功能为主，我们称为“处理器芯片”，有的芯片中的处理器只是辅助的功能，我们称之为“普通芯片”或“芯片”。

就像每一辆汽车都需要一台发动机，汽车生产商需要向其他的发动机生产商采购发动机一样。每一款芯片都需要一个或者多个处理器，因此“高通”“联发科”“三星”和“德州仪器”等芯片生产商需要采购处理器，于是他们可以从 ARM 公司采购处理器。

- 所谓“ARM 架构”就好像是发动机的设计图样一样，是由 ARM 公司发明并专利保护的“处理器架构”，ARM 公司基于此架构设计的处理器便是“ARM 架构处理器”或“ARM 处理器”。由于 ARM 主要以 IP 的形式授权其处理器，因此常称为“ARM 处理器 IP”。
- 通过直接授权“ARM 处理器 IP”给其他的芯片生产商（合作伙伴），这便是 ARM 公司的主要盈利模式。

芯片公司每设计一款芯片，如果购买了 ARM 公司提供的“ARM 处理器 IP”，芯片公司需要支付一笔前期授权费（Upfront License Fee），之后，如果该芯片被大规模生产销售，则每卖出一片芯片均需要按其售价向 ARM 公司支付一定比例（譬如售价的 1%～2%）的版税（Royalty Fee）。

由于 ARM 架构占据了绝大多数的市场份额，形成了完整的软件生态环境，在移动和嵌入式领域的芯片厂商，购买 ARM 处理器 IP 几乎成为这些厂商的首选。

就像有些有实力的汽车生产商可以自己设计制造发动机一样。有实力的芯片公司也想自己设计处理器，因此有 3 个选择。

- 第 1 个选择：自己发明一种处理器架构。
- 第 2 个选择：购买其他商业公司的“非 ARM 架构”处理器 IP。
- 第 3 个选择：购买 ARM 公司的“ARM 架构授权”而不是直接购买“ARM 处理器 IP”，自己定制开发基于 ARM 架构的处理器。

在前面的章节，我们曾经阐述了处理器架构及其衍生出的软件生态环境的重要性，探讨了为什么“非 ARM 架构”无法取得如 ARM 般巨大的成功，因此上述“第 1 个选择”和“第 2 个选择”在 ARM 架构占主导（譬如移动手持设备）的领域具有极大的风险。那么“第 3

个选择”便成为了这些有实力的芯片公司的几乎唯一选择。

- 就像汽车公司可以购买发动机公司的图样，然后按照自己的产品需求深度定制其发动机一样。芯片公司也可以通过购买 ARM 公司的“ARM 架构授权”，按照自己的产品需求深度定制其自己的处理器。
- 转让“ARM 架构授权”给其他的芯片生产商（合作伙伴），这便是 ARM 公司的另外一种盈利模式。

使用这种自主研发处理器的芯片在大规模生产销售后无需向 ARM 公司逐片支付版税，从而达到降低产品成本和提高产品差异性的效果。

只有实力最为雄厚的芯片公司才具备购买“ARM 架构授权”的能力。首先，因为 ARM 架构授权价格极其昂贵（高达千万美元量级），远远高于直接购买“ARM 处理器 IP”所需的前期授权费；其次，深度定制其自研处理器需要解决极高的技术难度与投入高昂的研发成本。目前有能力坚持做到这一点的也仅有“苹果”“高通”“华为”等巨头。

综上可以看出，“ARM 架构处理器”可以分为两种。

- 由 ARM 公司开发并出售的 IP，也俗称为公版 ARM。
- 由芯片公司基于 ARM 架构授权自主开发的私有内核，也俗称为定制自研 ARM。

相对应的，ARM 公司的主要盈利模式也可以分为两种。

- 授权“ARM 处理器 IP”给其他的芯片生产商（合作伙伴），收取对应的前期授权费（Upfront License Fee），以及量产后的版税。
- 转让“ARM 架构授权”给其他的芯片生产商（合作伙伴），收取对应的架构授权费。

ARM 公司的强大之处便在于其与众多合作伙伴一起构建了强大的 ARM 阵营，如图 1-10 所示。全世界目前几乎大多数主流芯片公司都直接或者间接地在使用 ARM 架构处理器。



图 1-10　ARM 公司合作伙伴图谱

ARM 公司自 2004 年推出 ARMv7 内核架构时，便摒弃了以往“ARM+数字”这种处理器命名方法（之前的处理器统称经典处理器系列），启用 Cortex 来命名，并将 Cortex 系列细分为三大类，如图 1-11 所示。

- Cortex-A：面向性能密集型系统的应用处理器核。
- Cortex-R：面向实时应用的高性能核。
- Cortex-M：面向各类嵌入式应用的微控制器核。

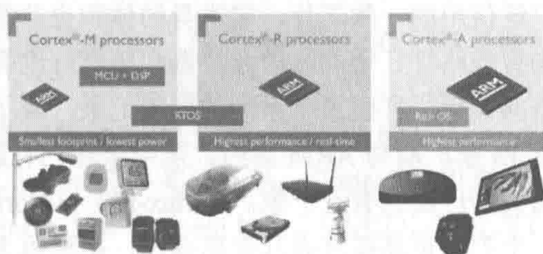


图 1-11　Cortex-A/R/M 系列

其中，Cortex-A 系列与 Cortex-M 系列的成功尤其引人瞩目。接下来的章节将对 Cortex-M 系列与 Cortex-A 系列的成功分别加以详细论述。

1.4.2 小个子有大力量——无处不在的 Cortex-M 系列

Cortex-M 是一组用于低功耗微控制器领域的 32 位 RISC 处理器系列，包括 Cortex-M0、Cortex-M0+、Cortex-M1、Cortex-M3、Cortex-M4(F)、Cortex-M7(F)、Cortex-M23 和 Cortex-M33(F)。如果 Cortex-M4 / M7 / M33 处理器包含了硬件浮点运算单元（FPU），也称为 Cortex-M4F/M7F/M33F。表 1-3 列出了 Cortex-M 系列各处理器的发布时间和特点。

表 1-3　ARM Cortex-M 系列各处理器发布时间和特点

型　号	发布时间	流水线深度	描述
Cortex-M3	2004	3 级	面向标准嵌入式市场的高性能低成本的 ARM 处理器
Cortex-M1	2007	3 级	专门面向 FPGA 中设计实现的 ARM 处理器
Cortex-M0	2009	3 级	面积最小和能耗极低的 ARM 处理器
Cortex-M4	2010	3 级	在 M3 基础上增加单精度浮点、DSP 功能以满足数字信号控制市场的 ARM 处理器
Cortex-M0+	2012	2 级	在 M0 基础上进一步降低功耗的 ARM 处理器
Cortex-M7	2014	6 级	超标量设计，配备分支预测单元，不仅支持单精度浮点，还增加了硬件双精度浮点能力，进一步提升计算性能和 DSP 处理能力，主要面向高端嵌入式市场
Cortex-M23	2016	2 级	可以简单理解为在 Cortex-M0+的基础上增加了硬件整数除法器与安全特性（TrustZone Security）
Cortex-M33	2016	3 级	可以简单理解为在 Cortex-M4 的基础上增加了安全特性（TrustZone Security）

Cortex-M 的应用场景虽然不像 Cortex-A 系列那样光芒四射，但是其应用的嵌入式领域需求量巨大。有数据显示，2018 年物联网设备的数量将会超过移动设备，到了 2021 年，我

们将会拥有 18 亿台 PC、86 亿台移动设备和 157 亿台物联网设备。譬如有一些物联网设备可能需要在几年的时间里运转，而且仅依靠自身所带的电池，Cortex-M0 由于其体积非常之小而且功耗极低，就非常适合这类产品，比如传感器。而 Cortex-M3 是 Cortex 产品家族中最为广泛使用的一款芯片，它本身的体积也非常小，可以广泛应用于各种各样嵌入智能设备，比如智能路灯、智能家居温控器和智能灯泡等。2009 年 Cortex-M0 这款超低功耗的 32 位处理器问世后，打破了一系列的授权记录，成了各制造商竞相争夺的香饽饽，仅 9 个月时间，就有 15 家厂商与 ARM 签约。至今全球已有超过 60 家公司获得了 ARM Cortex-M 的授权，中国大陆厂商也有近十家。Cortex-M3 与 Cortex-M0 的合计出货量已经超过 200 亿片，其中有一半的出货是在过去几年完成的，据称每 30 分钟的出货量就可以达到 25 万片。

Cortex-M 另一个取得巨大成功的领域便是微控制器（Microcontroller Unit，MCU）。随着越来越多的电子厂商不断为物联网（IoT）推出新产品，全球微控制器（MCU）市场出货量出现巨大成长动能，且呈现出量价齐升的情况。据市场调研机构预测，2016~2020 年全球微控制器（MCU）出货量与销售额将持续创新高。

在 ARM 推出 Cortex-M 之前，全球主要的几个 MCU 芯片公司大多采用 8 位、16 位内核或者其自有的 32 位架构的处理器。ARM 推出 Cortex-M 处理器之后，迅速受到市场青睐，一些主流 MCU 供应商开始选择这款内核生产 MCU。

- 2007 年 6 月，ST 推出基于 ARM Cortex-M3 处理器核的 STM32 F1 系列 MCU 使之大放光芒。
- 2009 年 3 月，恩智浦半导体 NXP 率先推出了第一款基于 ARM Cortex-M0 处理器的 LPC1100 系列 MCU。
- 2010 年 8 月，飞思卡尔半导体 Freescale（2015 年被 NXP 并购）率先推出了第一款基于 ARM Cortex-M4 处理器的 Kinetis K 系列 MCU。
- 2012 年 11 月，恩智浦半导体 NXP 继续率先推出了第一款基于 ARM Cortex-M0+处理器的 LPC800 系列 MCU。
- 2014 年 9 月，意法半导体 ST 率先推出了第一款基于 ARM Cortex-M7 处理器的 STM32 F7 系列 MCU。

各家供应商采用 Cortex-M 处理器核加之以自己特别的开发，在市场中提供差异化的 MCU 产品，有些产品专注最佳能效、最高性能，而有些产品则专门应用于某些细分市场。

至今，主要的 MCU 厂商中几乎都有使用 ARM 的 Cortex-M 内核的产品线。可以肯定地说，Cortex-M 之于 32 位 MCU 就如同 8051（受到众多供应商支持的工业标准内核）之于 8 位 MCU。未来 Cortex-M 系列的 MCU 产品替代传统的 8051 或其他专用架构是大势所趋。甚至有声音表示：“未来，MCU 产品将不再按 8 位，16 位和 32 位来分，而是会按照 M0 核，M3 核以及 M4 核等 ARM 内核的种类来分。”作者不得不替非 ARM 架构的商业处理器厂商

们拊膺长叹："既生瑜，何生亮啊。"

1.4.3 移动王者——Cortex-A 系列在手持设备领域的巨大成功

Cortex-A 是一组用于高性能低功耗应用处理器领域的 32 位和 64 位 RISC 处理器系列。32 位架构的处理器包括 Cortex-A5、Cortex-A7、Cortex-A8、Cortex-A9、Cortex-A12、Cortex-A15、Cortex-A17 和 Cortex-A32。64 位架构的包括 ARM Cortex-A35、ARM Cortex-A53、ARM Cortex-A57、ARM Cortex-A72 和 ARM Cortex-A73。Cortex-A、Cortex-M 和 Cortex-R 架构的最大区别是包含了存储器管理单元（Memory Management Unit，MMU），因此可以支持操作系统的运行。

ARM 在 2005 年向市场推出 Cortex-A8 处理器，是第一款支持 ARMv7-A 架构的处理器。在当时的主流工艺下，Cortex-A8 处理器的速率可以在 600M～1GHz 的范围调节，能够满足那些需要工作在 300mW 以下的功耗优化的移动设备的要求，以及满足那些需要 2000 Dhrystone MIPS 的性能优化的消费类应用的要求。当 Cortex-A8 在 2008 年投入批量生产时，高带宽无线连接（3G）已经问世，大屏幕也用于移动设备，Cortex-A8 芯片的推出正好赶上了智能手机大发展的滥觞。

推出 Cortex-A8 之后不久，ARM 又推出了首款支持 ARMv7-A 架构的多核处理器 Cortex-A9。Cortex-A9 利用硬件模块来管理 CPU 集群中 1～4 个核的高速缓存一致性，加入了一个外部二级高速缓存。在 2011 年底和 2012 年初，当移动 SoC 设计人员可以采用多个核之后，性能得到进一步提升。旗舰级高端智能手机迅速切换到 4 核 Cortex-A9。除了开启了多核性能大门之外，与 Cortex-A8 相比，每个 Cortex-A9 处理器的单时钟周期指令吞吐量提高了大约 25%。这个性能的提升是在保持相似功耗和芯片面积的前提下，通过缩短流水线并乱序执行，以及在流水线早期阶段集成 NEON SIMD 和浮点功能而实现的。

如果说 Cortex-A8 牛刀小试让 ARM 初尝甜头，那么 Cortex-A9 则催生了智能手机的井喷期，Cortex-A9 几乎成了当时智能手机的标配，大量的智能手机采用了该内核，ARM 为此挣了个盆满钵盈。自此，ARM 便开始了它开挂的"下饺子"模式，以平均每年一款或多款的速度疯狂推出各款不同的 Cortex-A 处理器，迅速拉开与竞争对手的差距。具体 ARM Cortex-A 系列各处理器的发布时间和特点，见表 1-4。

表 1-4　ARM Cortex-A 系列各处理器发布时间和特点

型　号	发布年份	位数	架构	流水线深度	指令发射类型	乱序执行	核数
Cortex-A8	2005	32	ARMv7-A	13 级	双发射	乱序执行	1
Cortex-A9	2007	32	ARMv7-A	8 级	双发射	乱序执行	1～4
Cortex-A5	2009	32	ARMv7-A	8 级	单发射	顺序执行	1～4
Cortex-A15	2010	32	ARMv7-A	15 级	三发射	乱序执行	1～4

续表

型　号	发布年份	位数	架构	流水线深度	指令发射类型	乱序执行	核数
Cortex-A7	2011	32	ARMv7-A	8 级	部分双发射	顺序执行	1～8
Cortex-A53	2011	64	ARMv8-A	可以理解为 A7 的 64 位版			
Cortex-A57	2010	64	ARMv8-A	可以理解为 A15 的 64 位版			
Cortex-A12	2013	32	ARMv7-A	可以理解为 A9 的性能提升优化版本			
Cortex-A17	2014	32	ARMv7-A	可以理解为 A12 的进一步性能提升，优化版本			
Cortex-A35	2015	64	ARMv8-A	8 级	部分双发射	顺序执行	1～8
Cortex-A72	2015	64	ARMv8-A	可以理解为 A57 的性能提升优化版本			
Cortex-A73	2015	64	ARMv8-A	可以理解为 A72 的性能进一步提升优化版本			
Cortex-A32	2016	32	ARMv8-A	可以理解为 A35 的 32 位版本			
Cortex-A55	2017	64	ARMv8.2-A	可以理解为 A53 的功耗进一步提升优化版本			
Cortex-A75	2017	64	ARMv8.2-A	可以理解为 A73 的性能进一步提升优化版本			

ARM 推出 Cortex-A 系列各款处理器的速度之快、之多，显示了其研发机器的超强生产力，由于其推出的处理器型号太多、太快，阿拉伯数字都不够用了，如表 1-4 所示，其型号的编号规则逐渐令作者都傻傻分不清了。同时，由于其推出的处理器型号太多、太快，乃至于令众多授权 ARMv7/8-A 架构进行自研处理器的巨头都疲于奔命。在 Cortex-A8/A9 时代，多家有实力的巨头均选择授权 ARMv7/8-A 架构进行自研处理器以差异化其产品并降低成本。这些巨头包括高通、苹果、Marvell、博通、三星、TI 以及 LG 等。作者便曾经在其中的一家巨头供职担任 CPU 高级设计工程师，开发其自研的 Cortex-A 系列高性能处理器。如前所述，研发一款高性能的应用处理器需要解决挑战极高的技术难题以及投入数年时间，而当 ARM 以年均一款新品之势席卷市场之时，使得自研处理器没有能够来得及推出便已过时。众巨头们纷纷弃甲丢盔，相继有 TI、博通、Marvell 和 LG 等巨头放弃了自研处理器业务。乃至自研处理器做的最为成功的高通（以其 Snapdragon 系列应用处理器风靡市场）也在其中低端 SoC 产品中放弃了自研处理器转而采购 ARM 的 Cortex-A 系列处理器，仅在高端 SoC 中保留了自研的处理器。值得一提的是，由于中国的巨大市场与产业支持，在巨头们放弃自研处理器的趋势下，中国的手机巨头华为与展讯逆势而上，开始授权 ARMv8-A 架构进行自研处理器的研发，并取得了令人欣喜的成果。

Cortex-A 系列的巨大成功彻底地奠定了 ARM 在移动领域的统治地位。由于 Cortex-A 系列的先机与成功，ARM 架构在移动领域构筑了城宽池阔的软件生态环境。至今，ARM 架构已经应用到全球 85%的智能移动设备中，其中有超过 95%的智能手机都基于 ARM 的设计，基本上使得其他架构的处理器失去了进入该领域的可能性。ARM 携 Cortex-A 系列移动领域一统江山，就如坦格利安人驾着喷火的巨头征服维斯特洛大陆般如入无人之境。ARM 除了一步步提升 Cortex 架构性能之余，也找到了很多“志同道合”的伙伴，比如高通、谷歌和

微软等，并与合作伙伴们形成了强大的生态联盟。携此余威，传统 x86 架构的 PC 与服务器领域就成为了 ARM 的下一步发展目标。有道是“驱巨兽鼎定移动地，Cortex-A 剑指服务区”。预知后事如何，且听下节分解。

1.4.4 进击的巨人——ARM 进军 PC 与服务器领域的雄心

PC 与服务器市场是一个超千亿规模的大蛋糕，而这个市场长时间由另外一个巨头 Intel 把持，同为 x86 阵营的 AMD 常年屈居老二，分享着有限的蛋糕份额。Intel 在此领域的巨大成功，是其丰厚盈利的主要来源。

上一节提到 ARM 剑指 PC 与服务器领域，谷歌 ChromeBook 就是 ARM 挥师 PC 市场的先行军，在（海外的）入门级市场受到了广泛好评，ARM 处理器可以帮助此类设备变得更轻、更省电。微软对 ARM 的支持同样给力，2016 年 12 月举行的 WinHEC 2016 大会上，微软与高通宣布将采用下一代骁龙处理器（基于 ARM 架构）的移动计算终端上支持 Windows 10 系统，微软演示了搭载骁龙 820 处理器的笔记本运行 Windows 10，骁龙 820 在 4GB 存储器支撑下（性能可以和 Intel i3 媲美），一台 Windows 10 企业版系统笔记本能够流畅地运行 Edge、外接绘图板、观看高清视频、使用 PS 定向滤镜等，同时支持多任务后台。

2017 年，高通宣布正在对其自研骁龙 835 进行优化，将这款处理器扩展到运行 Windows 10 的移动 PC 当中，而搭载骁龙 835 的 Windows 10 移动 PC 计划在 2017 年第四季度推出。除此之外，在数据中心领域，高通也与微软达成了合作，未来运行 Windows Server 的服务器也可以搭载高通 10nm Centriq 处理器，这也是业内首款 10nm 服务器处理器。微软还宣布将在未来的 Windows 10 RedStone 3 当中正式提供 ARM 设备对完整版 Windows 10 的兼容支持，这意味着基于 ARM 处理器的设备可以运行 x86 程序，跨平台融合正式到来。

至此，我们已经介绍 ARM 公司及其 ARM 架构的强大之处，了解了 Cortex-M 处理器在嵌入式领域内的巨大成功，Cortex-A 处理器在移动领域内的王者之位，甚至于在 PC 与服务器领域内的雄心。

1.5 东边日出西边雨，道是无晴却有晴——RISC-V 登场

RISC-V 架构主要由美国加州大学伯克利分校（简称伯克利）的 Krste Asanovic 教授、Andrew Waterman 和 Yunsup Lee 等开发人员于 2010 年发明，并且得到了计算机体系结构领域的泰斗 David Patterson 的大力支持。伯克利的开发人员之所以发明一套新的指令集架构，而不是使用成熟的 x86 或者 ARM 架构，是因为这些架构经过多年的发展变得极为复杂和冗繁，并且存在着高昂的专利和架构授权问题。并且修改 ARM 处理器的 RTL 代码是不被支持

的，而 x86 处理器的源代码根本不可能获得到。其他的开源架构（譬如 SPARC、OpenRISC）均有着或多或少的问题（第 2 章将详细论述）。有感于计算机体系结构和指令集架构已经过数十年的发展非常成熟，但是像伯克利这样的研究机构竟然“无米下锅”（选择不出合适的指令集架构供其使用）。伯克利的教授与研发人员决定发明一种全新的、简单且开放免费的指令集架构，于是 RISC-V 架构诞生了。

有关 RISC-V 的诞生，有兴趣的读者可以自行到网络中查阅文章《伯克利希望将 RISC-V 开源架构推向主流》。

RISC-V（英文读作“risk-five”），是一种全新的指令集架构。“V”包含两层意思，一是这是 Berkeley 从 RISC I 开始设计的第五代指令集架构；二是它代表了变化（Variation）和向量（Vectors）。

经过几年的开发，伯克利为 RISC-V 架构开发除了完整的软件工具链以及若干开源的处理器实例，得到越来越多的人的关注。2016 年，RISC-V 基金会（Foundation）正式成立开始运作。RISC-V 基金会是一个非营利性的组织，负责维护标准的 RISC-V 指令集手册与架构文档，并推动 RISC-V 架构的发展。

RISC-V 架构的目标如下。

- 成为一种完全开放的指令集，可以被任何学术机构或商业组织所自由使用。
- 成为一种真正适合硬件实现且稳定的标准指令集。

RISC-V 基金会负责维护标准的 RISC-V 架构文档和编译器等 CPU 所需的软件工具链，任何组织和个人可以随时在 RISC-V 基金会网站上免费下载（无须注册）。

RISC-V 的推出以及基金会的成立，受到了学术界与工业界的巨大欢迎。著名的科技行业分析公司 Linley Group 将 RISC-V 评为“2016 年最佳技术”，如图 1-12 所示。



图 1-12　RISC-V 架构标志图

开放而免费的 RISC-V 架构诞生，不仅对于高校与研究机构是个好消息；为前期资金缺乏的创业公司、成本极其敏感的产品、对现有软件生态依赖不大的领域，都提供了另外一种选择，而且得到了业界主要科技公司的拥戴，包括谷歌、惠普、Oracle 和西部数据等硅谷巨头都是 RISC-V 基金会的创始会员，如图 1-13 所示。众多的芯片公司已经开始使用（譬如，三星、英伟达等）或者计划使用 RISC-V 开发其自有的处理器用于其产品。

RISC-V 基金会组织每年举行两次公开的专题讨论会（Workshop），以促进 RISC-V 阵营的交流与发展，任何组织和个人均可以从 RISC-V 基金会的网站上下载到每次 Workshop 上演示的 PPT 与文档。RISC-V 第六次 Workshop 于 2017 年 5 月在中国的上海交通大学举办，如图 1-14 所示，吸引了大批的中国公司和爱好者参与。



图 1-13 RISC-V 基金会创始会员，铂金、金、银级会员图谱

由于许多现在主流的计算机体系结构英文教材（譬如，计算机体系结构量化研究方法、计算机组成与设计等）的作者本身也是 RISC-V 架构的发起者，因此这些英文教材都相继推出了以 RISC-V 架构为基础的新版本教材，如图 1-15 所示。这意味着美国的大多数高校都将开始采用 RISC-V 作为教学范例，也意味着若干年后的高校毕业生都将对 RISC-V 架构非常熟知。



图 1-14 上海交通大学举办的 RISC-V 第六次 Workshop



图 1-15 经典教材计算机组成与设计最新版本

但是，一款指令集架构（ISA）最终能否取得成功，很大程度上取决于软件生态环境。罗马不是一天建成的，x86 与 ARM 架构经过多年的经营，构建了城宽池阔的软件生态环境，可以说是兵精粮足，非常强大。因此，作者认为 RISC-V 架构在短时间内还无法对 x86 和 ARM 架构形成撼动。但是随着越来越多的公司和项目开始采用 RISC-V 架构的处理器，相信 RISC-V 的软件生态也会逐步壮大起来。

本节虽然陈述了若干 RISC-V 蓬勃发展的具体案例，但是由于 RISC-V 阵营正在快速地向前发展，可能在本书成书之时，RISC-V 阵营又诞生了更加令人欣喜的案例，请读者自行查阅互联网更新见闻。

第 2 章将详细介绍 RISC-V 架构的技术细节。

1.6 原来你是这样的“薯片”——ARM 的免费计划

在第 1.5 节中我们提到，RISC-V 架构的特点是开放而且免费，并且成立了专门的基金会组织推动其发展，这是以前任何一种处理器架构都不曾有过的。这种新的模式是否会对现有的商业处理器架构形成冲击呢？ARM 与 Intel 这样的行业巨头商业公司是否会感到压力呢？不得不客观地说，RISC-V 基金会诞生的时间还很短暂，RISC-V 架构的生态目前还不够强大，远远没有到达威胁到 ARM 与 Intel 的程度，因此 ARM 与 Intel 并未在任何公开的场合对 RISC-V 发表过评价。

在第 1.4.1 节中我们已经提到过 ARM 的商业模式，芯片公司每设计一款芯片，如果购买了 ARM 公司提供的“ARM 处理器”，芯片公司需要支付一笔前期授权费。之后，如果该芯片被大规模生产销售，每卖出一片芯片均需要按其售价向 ARM 公司支付一定比例的版税。但是在 2017 年 6 月，ARM 宣布了 Cortex-M3 和 Cortex-M0 两款处理器的免前期授权费计划。这意味着自此之后 ARM Cortex-M 系列的两款处理器 M0 和 M3 均被免除了早期授权费，用户仅需在量产芯片后向 ARM 逐片支付版税即可。这对于广大使用 Cortex-M 处理器的芯片公司而言无疑是个好消息。

有评论表示，ARM 之所以这样做可能也是对目前如火如荼的开放 RISC-V 架构的一种阻击。当然，作者认为这只是某些看客们毫无根据的个人观点，真实性不具备任何可考性。不过无论如何，作者认为，有竞争、有活力的市场，总比一家独大的寡头垄断要有趣得多。

1.7 旧时王谢堂前燕，飞入寻常百姓家——你也可以设计自己的处理器

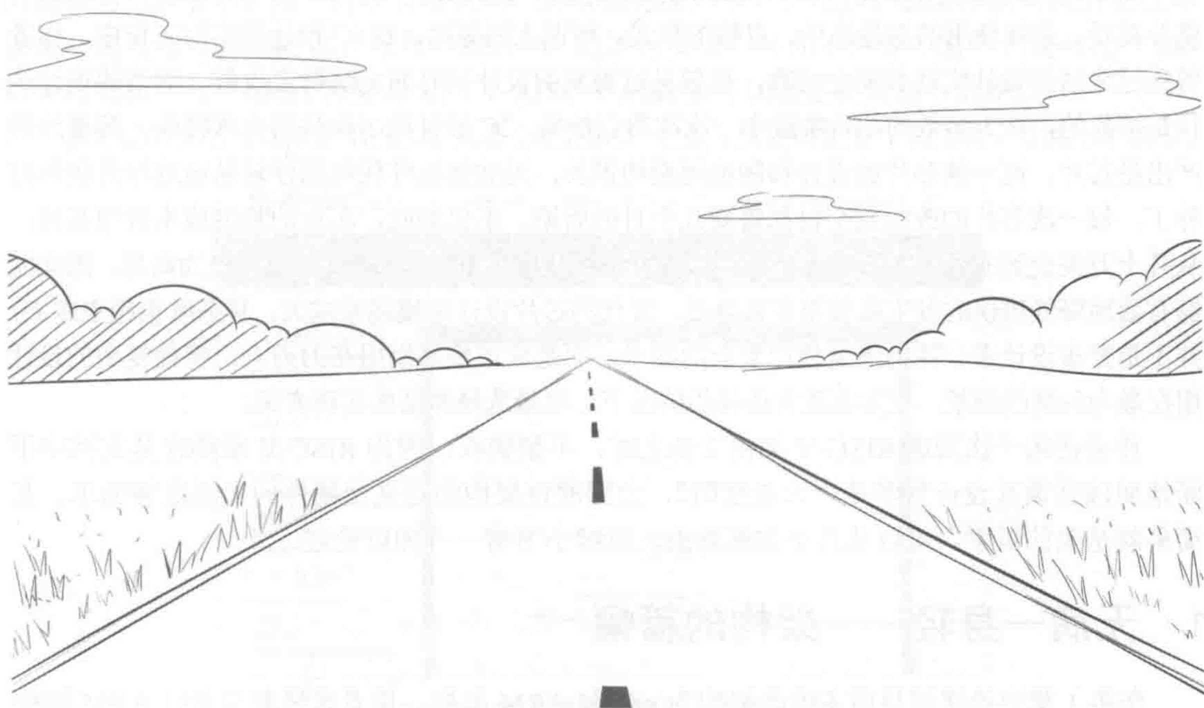
本章系统地论述了 CPU 的“三世三生”，也简述了 ARM 的如何强大以及开放 RISC-V 架构的诞生。

一言以蔽之，开放而免费 RISC-V 架构使得任何公司与个人均可受用，极大地降低了 CPU 设计的准入门槛。有了 RISC-V 架构，CPU 设计将不再是“权贵的游戏”，有道是“旧时王谢堂前燕，飞入寻常百姓家”——你也可以设计自己的处理器。

本书的第 2 章将详细介绍 RISC-V 架构的细节，在本书的第二部分将结合开源的蜂鸟 E200（基于 RISC-V 架构）实例详细介绍如何设计一款 RISC-V 处理器。

第2章 大道至简——RISC-V架构之魂

"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."
– Edsger W. Dijkstra



关于 RISC-V 架构的诞生初衷和背景，请参见第 1.5 节，本章在此不做重复赘述。本章将对 RISC-V 架构的设计思想进行深入浅出的介绍。

注意：本章中将会多次出现“RISC 处理器”“RISC 架构”“RISC-V 处理器”和“RISC-V 架构”等关键词。请初学者务必注意加以区别，如第 1 章中所述。

- RISC 表示精简指令集（Reduced Instruction Set Computer，RISC）。
- RISC-V 只是伯克利发明的一种特定指令集架构（属于 RISC 类型）。

2.1 简单就是美——RISC-V 架构的设计哲学

RISC-V 架构作为一种指令集架构，在介绍细节之前，让我们先了解设计的哲学。所谓设计的“哲学”便是其推崇的一种策略，譬如我们熟知的日本车的设计哲学是经济省油，美国车的设计哲学是霸气等。RISC-V 架构的设计哲学是什么呢？是“大道至简”。

作者最为推崇的一种设计哲学便是：简单就是美，简单便意味着可靠。无数的实际案例已经佐证了“简单即意味着可靠”的真理，反之越复杂的机器则越容易出错。一个最好的例子便是著名的 AK47 冲锋枪，正是由于简单可靠的设计哲学，使其性价比和可靠性极其出众，成为世界上应用最广泛的单兵武器。

在格斗界，初学者往往容易陷入追求花式繁复技巧的泥淖，迷信于花拳绣腿。然而顶级的格斗高手，最终使用的都是简单、直接的招式。所谓大道至简，在 IC 设计的实际工作中，作者曾见过简洁的设计实现其安全可靠，也曾见过繁复的设计长时间无法稳定收敛。简洁的设计往往是可靠的，在大多数的项目实践中一次次得到检验。IC 设计的工作性质非常特殊，其最终的产出是芯片，而一款芯片的设计和制造周期均很长，无法像软件代码那样轻易地进行升级和打补丁，每一次芯片的改版到交付都需要几个月的周期。不仅如此，芯片的制造成本费用高昂，从几十万美金到成百上千万美金不等。这些特性都决定了 IC 设计的试错成本极为高昂，因此能够有效地降低错误的发生就显得非常重要。现代的芯片设计规模越来越大，复杂度也越来越高，并不是要求设计者一味地逃避使用复杂的技术，而是应该将好钢用在刀刃上，将最复杂的设计用在最为关键的场景，在大多数有选择的情况下，尽量选择简洁的实现方案。

作者在第一次阅读 RISC-V 架构文档之时，不禁赞叹。因为 RISC-V 架构在其文档中不断地明确强调其设计哲学是“大道至简”，力图通过架构的定义使硬件的实现足够简单。其简单就是美的哲学，可以从几个方面看出，后续小节将一一加以论述。

2.1.1 无病一身轻——架构的篇幅

在第 1 章中论述过目前主流的架构为 x86 与 ARM 架构。作者曾经参与设计 ARM 架构

的应用处理器，因此需要阅读 ARM 的架构文档。如果对 ARM 的架构文档熟悉的读者应该了解其篇幅。经过几十年的发展，现在的 x86 与 ARM 架构的架构文档多达数千页，打印出来能有半个桌子高，可真是“著作等身”。

想必 x86 与 ARM 架构在诞生之初，其篇幅也不至于像现在这般长篇累牍。之所以架构文档长达数千页，且版本众多，一个主要的原因是其架构发展的过程也伴随了现代处理器架构技术的不断发展成熟，并且作为商用的架构，为了能够保持架构的向后兼容性，不得不保留许多过时的定义，或者在定义新的架构部分时为了能够兼容已经存在的技术部分而显得非常的别扭。久而久之就变成了老太婆的裹脚布——极为冗长，可以说是积重难返。

那么现代成熟的架构是否能够选择重新开始，重新定义一个简洁的架构呢？可以说是几乎不可能。Intel 也曾经在推出 Itanium 架构之时另起灶炉，放弃了向前兼容性，最终 Intel 的 Itanium 遭遇惨败，其中一个重要的原因便是其无法向前兼容，从而无法得到用户的接受。试想一下，如果我们买了一款具有新的处理器的计算机或者手机，之前所有的软件都无法运行，那肯定是无法让人接受的。

现在推出的 RISC-V 架构，则具备了后发优势。由于计算机体系结构经过多年的发展已经是一个比较成熟的技术，多年来在不断成熟的过程中暴露的问题都已经被研究透彻了，因此新的 RISC-V 架构能够加以规避，并且没有背负向后兼容的历史包袱，可以说是无病一身轻。

目前的“RISC-V 架构文档”分为“指令集文档”和“特权架构文档”。“指令集文档”的篇幅为 100 多页，而“特权架构文档”的篇幅也仅为 100 页左右。熟悉体系结构的工程师仅需一两天便可将其通读，虽然“RISC-V 的架构文档”还在不断地丰富，但是相比“x86 的架构文档”与“ARM 的架构文档”，RISC-V 的篇幅可以说是极其短小精悍。

感兴趣的读者可以登录 RISC-V 基金会的网站，无须注册便可免费下载文档，如图 2-1 所示。



图 2-1　RISC-V 基金会网站上的架构文档

2.1.2 能屈能伸——模块化的指令集

RISC-V 架构相比其他成熟的商业架构，最大的不同在于它是一个模块化的架构。因此 RISC-V 架构不仅短小精悍，而且其不同的部分还能以模块化的方式组织在一起，从而试图通过一套统一的架构满足各种不同的应用。

这种模块化是 x86 与 ARM 架构所不具备的。以 ARM 的架构为例，ARM 的架构分为 A、R 和 M，共 3 个系列，分别针对应用操作系统（Application）、实时（Real-Time）和嵌入式（Embedded）3 个领域，彼此之间并不兼容。但是模块化的 RISC-V 架构能够使得用户灵活地选择不同的模块进行组合，以满足不同的应用场景，可以说是“老少咸宜”。例如针对小面积、低功耗的嵌入式场景，用户可以选择 RV32IC 组合的指令集，仅使用机器模式（Machine Mode）；而针对高性能应用操作系统场景，则可以选择例如 RV32IMFDC 的指令集，使用机器模式（Machine Mode）与用户模式（User Mode）两种模式。

在第 2.2.1 节中将会介绍 RISC-V 指令集模块化特性的详情。

2.1.3 浓缩的都是精华——指令的数量

短小精悍的架构和模块化的哲学，使得 RISC-V 架构的指令数目非常简洁。基本的 RISC-V 指令数目仅有 40 多条，加上其他的模块化扩展指令总共几十条指令。图 2-2 是 RISC-V 指令集图卡，请参见附录 A 了解 RISC-V 指令集的详细信息。

图 2-2　RISC-V 指令集图卡

2.2 RISC-V 指令集架构简介

本章将对 RISC-V 的指令集架构多方面的特性进行简要介绍。

注意：本节仅对 RISC-V 的指令集架构的特点进行概述和横向比较。有关 RISC-V 指令集架构的详情，请参见附录 A。在本节进行横向比较的描述中涉及许多处理器设计的常识和背景知识，对于完全不了解 CPU 的初学者而言可能难以理解，请参见第二部分与第三部分中的各章节进行系统学习后再来理解本节的内容。

2.2.1 模块化的指令子集

RISC-V 的指令集使用模块化的方式进行组织，每一个模块使用一个英文字母来表示。RISC-V 最基本也是唯一强制要求实现的指令集部分是由 I 字母表示的基本整数指令子集。使用该整数指令子集，便能够实现完整的软件编译器。其他的指令子集部分均为可选的模块，具有代表性的模块包括 M/A/F/D/C，如表 2-1 所示。

表 2-1　RISC-V 的模块化指令集

基本指令集	指令数	描　述
RV32I	47	32 位地址空间与整数指令，支持 32 个通用整数寄存器
RV32E	47	RV32I 的子集，仅支持 16 个通用整数寄存器
RV64I	59	64 位地址空间与整数指令及一部分 32 位的整数指令
RV128I	71	128 位地址空间与整数指令及一部分 64 位和 32 位的指令
扩展指令集	指令数	描　述
M	8	整数乘法与除法指令
A	11	存储器原子（Atomic）操作指令和 Load-Reserved/Store-Conditional 指令
F	26	单精度（32 比特）浮点指令
D	26	双精度（64 比特）浮点指令，必须支持 F 扩展指令
C	46	压缩指令，指令长度为 16 位

以上模块的一个特定组合“IMAFD”，也被称为“通用”组合，用英文字母 G 表示。因此 RV32G 表示 RV32IMAFD，同理 RV64G 表示 RV64IMAFD。

为了提高代码密度，RISC-V 架构也提供可选的“压缩”指令子集，用英文字母 C 表示。压缩指令的指令编码长度为 16 比特，而普通的非压缩指令的长度为 32 比特。

为了进一步减少面积，RISC-V 架构还提供一种“嵌入式”架构，用英文字母 E 表示。该架构主要用于追求极低面积与功耗的深嵌入式场景。该架构仅需要支持 16 个通用整数寄

存器，而非嵌入式的普通架构则需要支持 32 个通用整数寄存器。

通过以上的模块化指令集，能够选择不同的组合来满足不同的应用。例如，追求小面积、低功耗的嵌入式场景可以选择使用 RV32EC 架构；而大型的 64 位架构则可以选择 RV64G。

除了上述模块，还有若干的模块如 L、B、P、V 和 T 等。目前这些扩展大多数还在不断完善和定义中，尚未最终确定，因此不做详细论述。

2.2.2 可配置的通用寄存器组

RISC-V 架构支持 32 位或者 64 位的架构，32 位架构由 RV32 表示，其每个通用寄存器的宽度为 32 比特；64 位架构由 RV64 表示，其每个通用寄存器的宽度为 64 比特。

RISC-V 架构的整数通用寄存器组，包含 32 个（I 架构）或者 16 个（E 架构）通用整数寄存器，其中整数寄存器 0 被预留为常数 0，其他的 31 个（I 架构）或者 15 个（E 架构）为普通的通用整数寄存器。

如果使用浮点模块（F 或者 D），则需要另外一个独立的浮点寄存器组，包含 32 个通用浮点寄存器。如果仅使用 F 模块的浮点指令子集，则每个通用浮点寄存器的宽度为 32 比特；如果使用了 D 模块的浮点指令子集，则每个通用浮点寄存器的宽度为 64 比特。

请参见附录 A4.1 节了解 RISC-V 架构通用寄存器组的细节。

2.2.3 规整的指令编码

在流水线中能够尽快地读取通用寄存器组，往往是处理器流水线设计的期望之一，这样可以提高处理器性能和优化时序。这个看似简单的道理在很多现存的商用 RISC 架构中都难以实现，因为经过多年反复修改不断添加新指令后，其指令编码中的寄存器索引位置变得非常凌乱，给译码器造成了负担。

得益于后发优势和总结了多年来处理器发展的经验，RISC-V 的指令集编码非常规整，指令所需的通用寄存器的索引（Index）都被放在固定的位置，如图 2-3 所示。因此指令译码器（Instruction Decoder）可以非常便捷地译码出寄存器索引，然后读取通用寄存器组（Register File，Regfile）。

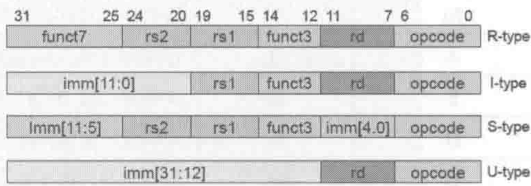


图 2-3 RV32I 规整的指令编码格式

请参见附录 F 了解 RISC-V 架构指令集列表和编码细节。

2.2.4 简洁的存储器访问指令

与所有的 RISC 处理器架构一样，RISC-V 架构使用专用的存储器读（Load）指令和存

储器写（Store）指令访问存储器（Memory），其他的普通指令无法访问存储器，这种架构是RISC 架构常用的一个基本策略。这种策略使得处理器核的硬件设计变得简单。存储器访问的基本单位是字节（Byte）。RISC-V 的存储器读和存储器写指令支持一个字节（8 位）、半字（16 位）、单字（32 位）为单位的存储器读写操作。如果是 64 位架构还可以支持一个双字（64 位）为单位的存储器读写操作。

RISC-V 架构的存储器访问指令还有如下显著特点。

- 为了提高存储器读写的性能，RISC-V 架构推荐使用地址对齐的存储器读写操作，但是也支持地址非对齐的存储器操作 RISC-V 架构。处理器既可以选择用硬件来支持，也可以选择用软件来支持。
- 由于现在的主流应用是小端格式（Little-Endian），RISC-V 架构仅支持小端格式。有关小端格式和大端格式的定义和区别，在此不做过多介绍。若对此不太了解的初学者可以自行查阅学习。
- 很多的 RISC 处理器都支持地址自增或者自减模式，这种自增或者自减的模式虽然能够提高处理器访问连续存储器地址区间的性能，但是也增加了设计处理器的难度。RISC-V 架构的存储器读和存储器写指令不支持地址自增自减的模式。
- RISC-V 架构采用松散存储器模型（Relaxed Memory Model），松散存储器模型对于访问不同地址的存储器读写指令的执行顺序不作要求，除非使用明确的存储器屏障（Fence）指令加以屏蔽。有关存储器模型（Memory Model）和存储器屏障指令的更多信息，请参见附录 A13。

这些选择都清楚地反映了 RISC-V 架构力图简化基本指令集，从而简化硬件设计的哲学。RISC-V 架构如此定义是具有合理性的，能达到能屈能伸的效果。例如，对于低功耗的简单 CPU，可以使用非常简单的硬件电路即可完成设计；而对于追求高性能的超标量处理器，则可以通过复杂设计的动态硬件调度能力来提高性能。

请参见附录 A14.2 节了解 RISC-V 架构存储器访问指令的细节。

2.2.5 高效的分支跳转指令

RISC-V 架构有两条无条件跳转指令（Unconditional Jump），即 jal 指令与 jalr 指令。跳转链接（Jump and Link）指令——jal 指令可用于进行子程序调用，同时将子程序返回地址存在链接寄存器（Link Register，由某一个通用整数寄存器担任）中。跳转链接寄存器（Jump and Link-Register）指令——jalr 指令能够用于子程序返回指令，通过将 jal 指令（跳转进入子程序）保存的链接寄存器用于 jalr 指令的基地址寄存器，则可以从子程序返回。请参见附录 A14.2 节了解 jal 和 jalr 指令的详细内容。

RISC-V 架构有 6 条带条件跳转指令（Conditional Branch），这种带条件的跳转指令跟普

通的运算指令一样直接使用两个整数操作数，然后对其进行比较。如果比较的条件满足，则进行跳转，因此此类指令将比较与跳转两个操作放在一条指令里完成。作为比较，很多其他的 RISC 架构的处理器需要使用两条独立的指令。第一条指令先使用比较指令，比较的结果被保存到状态寄存器之中；第二条指令使用跳转指令，判断前一条指令保存在状态寄存器当中的比较结果为真时，则进行跳转。相比而言，RISC-V 的这种带条件跳转指令不仅减少了指令的条数，同时硬件设计上更加简单。请参见附录 A14.2 节了解 6 条带条件跳转指令的详细内容。

对于没有配备硬件分支预测器的低端 CPU，为了保证其性能，RISC-V 的架构明确要求采用默认的静态分支预测机制，即如果是向后跳转的条件跳转指令，则预测为“跳”；如果是向前跳转的条件跳转指令，则预测为“不跳”，并且 RISC-V 架构要求编译器也按照这种默认的静态分支预测机制来编译生成汇编代码，从而让低端的 CPU 也得到不错的性能。

在低端的 CPU 中，为了使硬件设计尽量简单，RISC-V 架构特地定义了所有的带条件跳转指令跳转目标的偏移量（相对于当前指令的地址）都是有符号数，并且其符号位被编码在固定的位置。因此这种静态预测机制在硬件上非常容易实现，硬件译码器可以轻松地找到固定的位置，判断该位置的比特值为 1，表示负数（反之则为正数）。根据静态分支预测机制，如果是负数，则表示跳转的目标地址为当前地址减去偏移量，也就是向后跳转，则预测为“跳”。当然，对于配备有硬件分支预测器的高端 CPU，则还可以采用高级的动态分支预测机制来保证性能。

2.2.6 简洁的子程序调用

为了理解此节，需先对一般 RISC 架构中程序调用子函数的过程予以介绍，其过程如下。

- 进入子函数之后需要用存储器写（Store）指令来将当前的上下文（通用寄存器等的值）保存到系统存储器的堆栈区内，这个过程通常称为“保存现场”。
- 在退出子程序时，需要用存储器读（Load）指令来将之前保存的上下文（通用寄存器等的值）从系统存储器的堆栈区读出来，这个过程通常称为“恢复现场”。

“保存现场”和“恢复现场”的过程通常由编译器编译生成的指令完成，使用高层语言（例如 C 语言或者 C++）开发的开发者对此可以不用太关心。高层语言的程序中直接写上一个子函数调用即可，但是这个底层发生的“保存现场”和“恢复现场”的过程却是实实在在地发生着（可以从编译出的汇编语言里面看到那些“保存现场”和“恢复现场”的汇编指令），并且还需要消耗若干的 CPU 执行时间。

为了加速“保存现场”和“恢复现场”的过程，有的 RISC 架构发明了一次写多个寄存器到存储器中（Store Multiple），或者一次从存储器中读多个寄存器出来（Load Multiple）的指令。此类指令的好处是一条指令就可以完成很多事情，从而减少汇编指令的代码量，节省

代码的空间大小。但是“一次读多个寄存器指令”和“一次写多个寄存器指令”的弊端是会让 CPU 的硬件设计变得复杂，增加硬件的开销，也可能损伤时序，使得 CPU 的主频无法提高，作者曾经设计此类处理器时便深受其苦。

RISC-V 架构则放弃使用“一次读多个寄存器指令”和“一次写多个寄存器指令”。如果有的场合比较介意“保存现场”和“恢复现场”的指令条数，那么可以使用公用的程序库（专门用于保存和恢复现场）来进行，这样就可以省掉在每个子函数调用的过程中都放置数目不等的“保存现场”和“恢复现场”的指令。此选择再次印证了 RISC-V 追求硬件简单的哲学，因为放弃“一次读多个寄存器指令”和“一次写多个寄存器指令”可以大幅简化 CPU 的硬件设计，对于低功耗小面积的 CPU 可以选择非常简单的电路进行实现；而高性能超标量处理器由于硬件动态调度能力很强，可以有强大的分支预测电路保证 CPU 能够快速地跳转执行，从而可以选择使用公用的程序库（专门用于保存和恢复现场）的方式减少代码量，同时达到高性能。

2.2.7 无条件码执行

很多早期的 RISC 架构发明了带条件码的指令，例如在指令编码的头几位表示的是条件码（Conditional Code），只有该条件码对应的条件为真时，该指令才被真正执行。

这种将条件码编码到指令中的形式可以使编译器将短小的循环编译成带条件码的指令，而不用编译成分支跳转指令。这样便减少了分支跳转的出现，一方面减少了指令的数目；另一方面也避免了分支跳转带来的性能损失。然而，这种“条件码”指令的弊端同样会使 CPU 的硬件设计变得复杂，增加硬件的开销，也可能损伤时序使得 CPU 的主频无法提高。

RISC-V 架构则放弃使用这种带“条件码”指令的方式，对于任何的条件判断都使用普通的带条件分支跳转指令。此选择再次印证了 RISC-V 追求硬件简单的哲学，因为放弃带“条件码”指令的方式可以大幅简化 CPU 的硬件设计，对于低功耗小面积的 CPU 可以选择非常简单的电路进行实现，而高性能超标量处理器由于硬件动态调度能力很强，可以有强大的分支预测电路保证 CPU 能够快速地跳转执行达到高性能。

2.2.8 无分支延迟槽

很多早期的 RISC 架构均使用了“分支延迟槽（Delay Slot）”，具有代表性的便是 MIPS 架构。在很多经典的计算机体系结构教材中，均使用 MIPS 对分支延迟槽进行介绍。分支延迟槽就是指在每一条分支指令后面紧跟的一条或者若干条指令不受分支跳转的影响，不管分支是否跳转，这后面的几条指令都一定会被执行。请参见第 7.1.4 节了解更多分支延迟槽的背景知识。

早期的 RISC 架构很多采用了分支延迟槽诞生的原因主要是当时的处理器流水线比较简

单，没有使用高级的硬件动态分支预测器，使用分支延迟槽能够取得可观的性能效果。然而，这种分支延迟槽使得 CPU 的硬件设计变得极为别扭，CPU 设计人员对此苦不堪言。

RISC-V 架构则放弃了分支延迟槽，再次印证了 RISC-V 力图简化硬件的哲学，因为现代的高性能处理器的分支预测算法精度已经非常高，可以有强大的分支预测电路保证 CPU 能够准确地预测跳转执行达到高性能。而对于低功耗、小面积的 CPU，由于无须支持分支延迟槽，硬件得到极大简化，也能进一步减少功耗和提高时序。

2.2.9 零开销硬件循环

很多 RISC 架构还支持零开销硬件循环（Zero Overhead Hardware Loop）指令，其思想是通过硬件的直接参与，设置某些循环次数寄存器（Loop Count），然后可以让程序自动地进行循环，每一次循环则循环次数寄存器自动减 1，这样持续循环直到循环次数寄存器的值变成 0，则退出循环。

之所以提出发明这种硬件协助的零开销循环是因为在软件代码中的 for 循环（for i=0; i<N; i++）极为常见，而这种软件代码通过编译器编译之后，往往会编译成若干条加法指令和条件分支跳转指令，从而达到循环的效果。一方面这些加法和条件跳转指令占据了指令的条数，另一方面条件分支跳转存在分支预测的性能问题。而硬件协助的零开销循环，则将这些工作由硬件直接完成，省掉了加法和条件跳转指令，减少了指令条数且提高了性能。

然而，此类零开销硬件循环指令大幅地增加了硬件设计的复杂度。因此零开销循环指令与 RISC-V 架构简化硬件的哲学是完全相反的，在 RISC-V 架构中自然没有使用此类零开销硬件循环指令。

2.2.10 简洁的运算指令

在第 2.2.1 节中曾经提到 RISC-V 架构使用模块化的方式组织不同的指令子集，最基本的整数指令子集（I 字母表示）支持的运算包括加法、减法、移位、按位逻辑操作和比较操作。这些基本的运算操作能够通过组合或者函数库的方式完成更多的复杂操作（例如乘除法和浮点操作），从而完成大部分的软件操作。请参见附录 A14.2 节了解 RISC-V 架构整数运算指令的细节。

整数乘除法指令子集（M 字母表示）支持的运算包括有符号或者无符号的乘法和除法操作。乘法操作能够支持两个 32 位的整数相乘得到一个 64 位的结果；除法操作能够支持两个 32 位的整数相除得到一个 32 位的商与 32 位的余数。请参见附录 A14.3 节了解 RISC-V 架构整数乘法和除法指令的细节。单精度浮点指令子集（F 字母表示）与双精度浮点指令子集（D 字母表示）支持的运算包括浮点加减法、乘除法、乘累加、开平方根和比较等操作，同时提供整数与浮点、单精度与双精度浮点之间的格式转换操作。请参见附录 A14.4 节了解 RISC-V

架构浮点指令的细节。

很多 RISC 架构的处理器在运算指令产生错误之时，例如上溢（Overflow）、下溢（Underflow）、非规格化浮点数（Subnormal）和除零（Divide by Zero），都会产生软件异常。RISC-V 架构的一个特殊之处是对任何的运算指令错误（包括整数与浮点指令）均不产生异常，而是产生某个特殊的默认值，同时设置某些状态寄存器的状态位。RISC-V 架构推荐软件通过其他方法来找到这些错误。再次清楚地反映了 RISC-V 架构力图简化基本的指令集，从而简化硬件设计的哲学。

2.2.11 优雅的压缩指令子集

基本的 RISC-V 基本整数指令子集（字母 I 表示）规定的指令长度均为等长的 32 位，这种等长指令定义使得仅支持整数指令子集的基本 RISC-V CPU 非常容易设计。但是等长的 32 位编码指令也会造成代码体积（Code Size）相对较大的问题。

为了满足某些对于代码体积要求较高的场景（例如嵌入式领域），RISC-V 定义了一种可选的压缩（Compressed）指令子集，用字母 C 表示，也可以用 RVC 表示。RISC-V 具有后发优势，从一开始便规划了压缩指令，预留了足够的编码空间，16 位长指令与普通的 32 位长指令可以无缝自由地交织在一起，处理器也没有定义额外的状态。

RISC-V 压缩指令的另一个特别之处是，16 位指令的压缩策略是将一部分普通最常用的 32 位指令中的信息进行压缩重排得到（例如假设一条指令使用了两个同样的操作数索引，则可以省去其中一个索引的编码空间），因此每一条 16 位长的指令都能找到其一一对应的原始 32 位指令。因此程序编译成为压缩指令仅在汇编器阶段就可以完成，极大地简化了编译器工具链的负担。

RISC-V 架构的研究者进行了详细的代码体积分析，如图 2-4 所示，通过分析结果可以看出，RV32C 的代码体积相比 RV32 的代码体积减少了 40%，并且与 ARM、MIPS 和 x86 等架构相比有不错的表现。

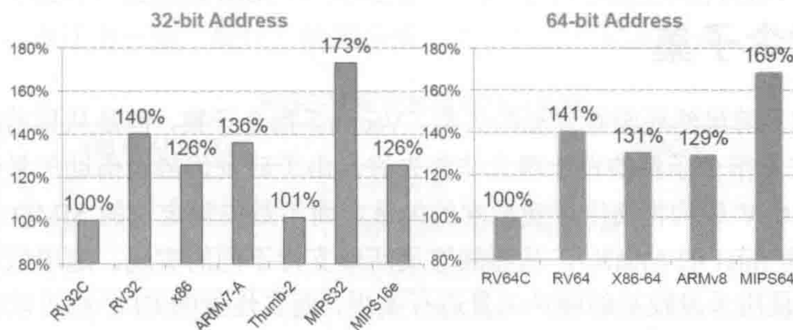


图 2-4 各指令集架构的代码密度比较（数据越小越好）

2.2.12 特权模式

RISC-V 架构定义了 3 种工作模式，又称为特权模式（Privileged Mode）。

- Machine Mode：机器模式，简称 M Mode。
- Supervisor Mode：监督模式，简称 S Mode。
- User Mode：用户模式，简称 U Mode。

RISC-V 架构定义 M Mode 为必选模式，另外两种为可选模式，通过不同的模式组合可以实现不同的系统。请参见附录 A8 了解更多工作模式的信息。

RISC-V 架构也支持几种不同的存储器地址管理机制，包括对于物理地址和虚拟地址的管理机制，使得 RISC-V 架构能够支持从简单的嵌入式系统（直接操作物理地址）到复杂的操作系统（直接操作虚拟地址）的各种系统。请参见附录 A12 了解更多信息。

2.2.13 CSR 寄存器

RISC-V 架构定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用自己的地址编码空间和存储器寻址的地址区间完全无关系。请参见附录 B 了解 CSR 寄存器的列表与详细信息。

CSR 寄存器的访问采用专用的 CSR 指令，包括 CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI 以及 CSRRCI 指令。请参见附录 A14.2 节了解相关指令的细节。

2.2.14 中断和异常

中断和异常机制往往是处理器指令集架构中最为复杂和关键的部分。RISC-V 架构定义了一套相对简单基本的中断和异常机制，但是也允许用户对其进行定制和扩展。请参见第 13 章系统地了解 RISC-V 中断和异常机制的详情。

2.2.15 矢量指令子集

RISC-V 架构目前虽然还没有定型的矢量（Vector）指令子集，但是从目前的草案中可以看出，RISC-V 矢量指令子集的设计理念非常先进。由于后发优势及借助矢量架构多年发展成熟的结论，RISC-V 架构将使用可变长度的矢量，而不是矢量定长的 SIMD 指令集（例如 ARM 的 NEON 和 Intel 的 MMX），从而能够灵活地支持不同的实现。追求低功耗、小面积的 CPU 可以选择使用长度较短的硬件矢量进行实现，而高性能的 CPU 则可以选择较长的硬件矢量进行实现，并且同样的软件代码能够互相兼容。

结合当前人工智能和高性能计算的强烈需求，一种开放开源矢量指令集的出现，倘若能够得到大量的开源算法软件库的支持，必将对产业界产生非常积极的影响，作者对此非常期待。

2.2.16 自定制指令扩展

除了上述阐述的模块化指令子集的可扩展、可选择，RISC-V 架构还有一个非常重要的特性，那就是支持第三方的扩展。用户可以扩展自己的指令子集，RISC-V 预留了大量的指令编码空间用于用户的自定义扩展，同时还定义了 4 条 Custom 指令可供用户直接使用。每条 Custom 指令都预留了几个比特位的子编码空间，因此用户可以直接使用 4 条 Custom 指令扩展出几十条自定义的指令。

请参见第 16 章了解更多协处理器扩展的信息。

2.2.17 总结与比较

处理器设计技术经过几十年的演进，随着大规模集成电路设计技术的发展直至今天，呈现如下特点。

- 由于高性能处理器的硬件调度能力已经非常强劲且主频很高，因此硬件设计希望指令集尽可能地规整、简单，从而使得处理器可以设计出更高的主频与更低的面积。
- 以 IoT 应用为主的极低功耗处理器更加苛求低功耗与低面积。
- 存储器的资源也比早期的 RISC 处理器更加丰富。

以上种种因素，使得很多早期的 RISC 架构设计理念（依据当时技术背景而诞生），不但不能帮助现代处理器设计，反而成了负担。某些早期 RISC 架构定义的特性，一方面使得高性能处理器的硬件设计束手束脚；另一方面又使得极低功耗的处理器硬件设计背负不必要的复杂度。

得益于后发优势，全新的 RISC-V 架构能够规避所有这些已知的负担，同时，利用其先进的设计哲学，设计出一套“现代”的指令集。本节再次总结其特点，如表 2-2 所示。

表 2-2　RISC-V 指令集架构特点总结

特　　性	x86 或 ARM 架构	RISC-V
架构篇幅	数千页	少于 300 页
模块化	不支持	支持模块化可配置的指令子集
可扩展性	不支持	支持可扩展定制指令
指令数目	指令数繁多，不同的架构分支彼此不兼容	一套指令集支持所有架构。基本指令子集仅 40 余条指令，以此为共有基础，加上其他常用模块子集指令总指令数也仅几十条

续表

特　性	x86 或 ARM 架构	RISC-V
易实现性	硬件实现得复杂度高	硬件设计与编译器实现非常简单 • 仅支持小端格式 • 存储器访问指令一次只访问一个元素 • 去除存储器访问指令的地址自增自减模式 • 规整的指令编码格式 • 简化的分支跳转指令与静态预测机制 • 不使用分支延迟槽（Delay Slot） • 不使用指令条件码（Conditional Code） • 运算指令的结果不产生异常（Exception） • 16 位的压缩指令有其对应的普通 32 位指令 • 不使用零开销硬件循环

RISC-V 的特点在于极简、模块化以及可定制扩展，通过这些指令集的组合或者扩展，几乎可以构建适用于任何一个领域的微处理器，比如云计算、存储、并行计算、虚拟化/容器、MCU、应用处理器和 DSP 处理器等。

2.3 RISC-V 软件工具链

在本章的介绍中已反复提到软件生态对于 CPU 的重要性，是运行于 CPU 之上的软件赋予了 CPU 以生命与灵魂，而软件工具链的完备则是 CPU 能够真正运行的第一步。

作为一种开放免费的架构，RISC-V 的软件工具链由开源社区维护，所有的工具链源代码均公开。可以通过 RISC-V 基金会网站进入 RISC-V Tools，如图 2-5 所示。

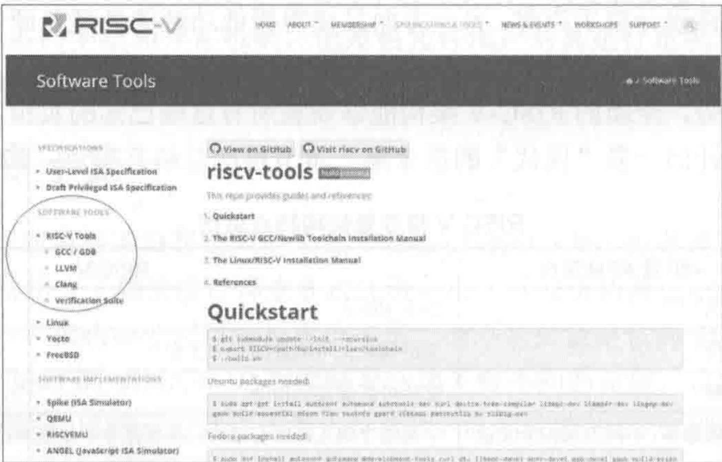


图 2-5　基金会网站 RISC-V Tools 页面

riscv-tools 的源代码在 GitHub 上被维护成一个宏项目（详情请在 GitHub 中搜索“riscv-tools”），其包含了所有 RISC-V 相关工具链、仿真器和测试套件等子项目，如图 2-6 所示。

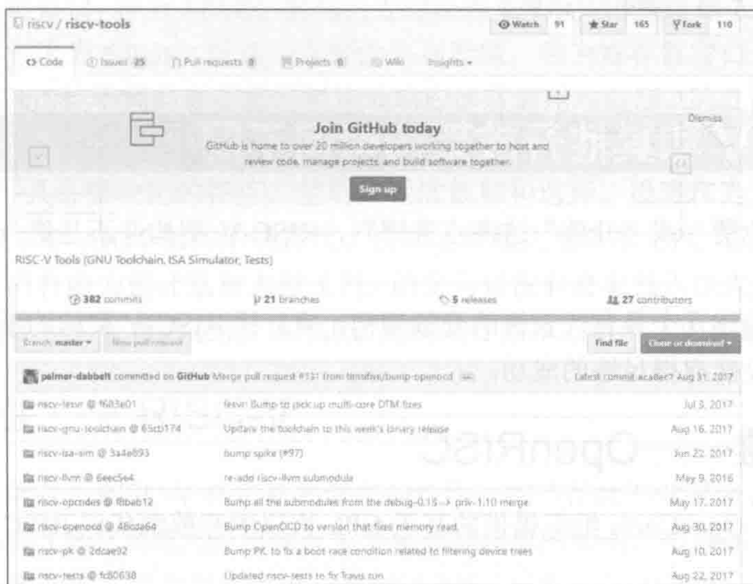


图 2-6　GitHub 上的 riscv-tools 项目

（1）riscv-fesvr 是一个用于实现上位机和 CPU 之间通信机制的库；riscv-pk 提供 RISC-V 可执行文件运行的程序运行环境，同时提供最简单的 bootloader；riscv-isa-sim 是一个基于 C/C++ 开发的指令集模拟器，其还有一个更通俗和为人所熟知的名字“Spike”。riscv-fesvr、riscv-pk 和 riscv-isa-sim 这 3 个工具协作在一起，可以用于在 Spike 模拟器上运行一个完整的程序。

（2）riscv-gnu-toolchain 是支持 RISC-V 的 GNU 工具链，包含了以下内容。

- riscv-gcc：GCC 编译器。
- riscv-binutils-gdb：二进制工具（链接器，汇编器等）、GDB 调试工具等。
- riscv-glibc：GNU C 标准库实现。

有关 GNU 工具链的更多信息请读者自行查阅。

- riscv-llvm 是一个基于 LLVM 编译器的框架，有关 LLVM 的更多信息请读者自行查阅。
- riscv-openocd 是一个基于 OpenOCD 的 RISC-V 调试器（Debugger）软件，请参见第 19.4 节了解更多 OpenOCD 的信息。
- riscv-opcodes 是一个 RISC-V 操作码信息转换脚本。
- riscv-tests 是一组 RISC-V 指令集测试用例。
- riscv-qemu 是一个支持 RISC-V 的 QEMU 模拟器，有关 QEMU 模拟器的更多信息请读者自行查阅。

如需使用RISC-V的工具链，除了按照GitHub上的说明下载源代码进行编译生成之外，还可以在网络上直接下载已经预先编译好的GNU工具链和Windows IDE开发工具，请参见第19.3节和第19.5节了解相关信息。

2.4 RISC-V和其他开放架构有何不同

如果仅从“免费”或“开放”这两点来评判，RISC-V架构并不是第一个做到免费或开放的处理器架构。

下面将通过论述几个具有代表性的开放架构，来分析RISC-V架构的不同之处以及为什么其他开放架构没能取得足够的成功。

2.4.1 平民英雄——OpenRISC

OpenRISC是OpenCores组织提供的基于GPL协议的开放源代码RISC处理器，它具有以下特点。

- 采用免费开放的32/64位 RISC架构。
- 用Verilog HDL（硬件描述语言）实现了基于该架构的处理器源代码。
- 具有完整的工具链。

OpenRISC被应用到很多公司的项目之中，可以说，OpenRISC是应用非常广泛的一种开源处理器实现。

OpenRISC的不足之处在于其侧重实现一种开源的CPU Core，而非立足于定义一种开放的指令集架构，因此其架构的发展不够完整。指令集的定义也不具备上节中提到的RISC-V架构的优点，更加没有上升到成立专门的基金会组织的高度。OpenRISC更多的时候被认为是一个开源的处理器核，而非一种优美的指令集架构。此外，OpenRISC的许可证为GPL，这意味着所有的指令集改动都必须开源（而RISC-V则无此约束）。

2.4.2 豪门显贵——SPARC

在第1章中已介绍过SPARC架构，作为经典的RISC微处理器架构之一，SPARC于1985年由Sun公司所设计。SPARC也是SPARC国际公司的注册商标之一，SPARC公司于1989年成立，目的是向外界推广SPARC架构以及为该架构进行兼容性测试。该公司为了推广SPARC的生态系统，将标准开放，并授权予多家生产商使用，包括德州仪器、Cypress半导体和富士通等。由于SPARC架构也对外完全开放，因此也出现了完全开放源码的LEON处理器（参见第1.1.4节的介绍）。不仅如此，Sun公司还于1994年推动SPARC v8架构成为

IEEE 标准（IEEE Standard 1754-1994）。

在第 1 章中介绍过，由于 SPARC 架构的初衷是面向服务器领域，其最大的特点是拥有一个大型的寄存器窗口，符合 SPARC 架构的处理器需要实现从 72 到 640 个之多的通用寄存器，每个寄存器宽度为 64bits，组成一系列的寄存器组，称为寄存器窗口。这种寄存器窗口的架构，由于可以切换不同的寄存器组快速地响应函数调用与返回，因此能够产生非常高的性能，但是这种架构由于功耗面积代价太大，而并不适用于 PC 与嵌入式领域处理器。而 SPARC 架构也不具备模块化的特点，使用户无法裁剪和选择。很难作为一种通用的处理器架构对商用的 x86 和 ARM 架构形成替代。设计这种超大服务器 CPU 芯片又非普通公司与个人所能完成，而有能力设计这种大型 CPU 的公司也没有必要投入巨大的成本来挑战 x86 的统治地位。随着 Sun 公司的衰弱，SPARC 架构现在基本上退出了人们的视野。

2.4.3 名校优生——RISC-V

在第 1.5 节介绍了 RISC-V 在伯克利诞生的经历，本节在此不做赘述。

因为多年来在 CPU 领域已经出现过多个免费或开放的架构，很多高校也在科研项目中推出过多种指令集架构。因此当作者第一次听说 RISC-V 时，以为又是一个玩具，或纯粹学术性质的科研项目而不以为意。

直到作者通读了 RISC-V 的架构文档，不禁为其先进的设计理念所折服。同时，RISC-V 架构的各种优点也得到了众多专业人士的青睐、好评和众多商业公司的相继加盟。并且 2016 年 RISC-V 基金会的正式启动在业界引起了不小的影响。如此种种，使得 RISC-V 成为至今为止最具革命性意义的开放处理器架构。

有兴趣的读者可以自行到网络中查阅文章《RISC-V 登场，Intel 和 ARM 会怕吗》《直指移动芯片市场，开源的处理器指令集架构发布》和《三星开发 RISC-V 架构自主 CPU 内核》。

第 3 章 乱花渐欲迷人眼——盘点 RISC-V 商业版本与开源版本



在第 1.5 节和第 2 章介绍了 RISC-V 架构的诞生和特点，注意：RISC-V 是一种开放的指令集架构，而不是一款具体的处理器。任何组织与个人均可以依据 RISC-V 架构设计实现自己的处理器，或是高性能处理器，抑或是低功耗处理器。只要是依据 RISC-V 架构而设计的处理器，都可以称为 RISC-V 架构处理器。

自从 RISC-V 架构诞生以来，在全世界范围内已经出现了数十个版本的 RISC-V 架构处理器，有的是开源免费的，有的是商业公司私有开发用于内部项目的，还有的是商业 IP 公司开发的 RISC-V 处理器 IP。本章将挑选几款比较知名开源免费 RISC-V 处理器（或 SoC）和商业公司开发的 RISC-V 处理器 IP，一一加以简述。

由于基于开放 RISC-V 架构的处理器在不断涌现，待本书成书之时，有可能已经出现了更多知名的 RISC-V 处理器，因此本书难免有信息不足之处，请读者自行查阅互联网更新。

3.1 各商业版本与开源版本综述

注意：本节对不同处理器版本的介绍中将使用许多处理器的关键特性参数或名称，对于完全不了解 CPU 的初学者而言可能难以理解，请参见本书第二部分与第三部分中的各章节进行系统学习后，再行理解本节。

3.1.1 Rocket Core （开源）

Rocket Core 是伯克利开发的一款开源 RISC-V 处理器核，可以由伯克利开发的 SoC 生成器 Rocket-Chip 生成。注意区分 Rocket Core 与 Rocket-Chip，Rocket Core 是一款处理器核；Rocket-Chip 是一款 SoC 生成器，用于生成若干伯克利开发的处理器核，包括 Rocket Core 和 BOOM Core。本节主要介绍 Rocket Core，而 BOOM Core 将在第 3.1.2 节介绍。

Rocket Core 是一款 64 位的处理器，结构如图 3-1 所示，具有如下特点。

- 具备可配置性，支持多种 RISC-V 的指令集扩展组合。
- 按序发射按序执行的五级流水线。
- 配备完整的指令 Cache 和数据 Cache。
- 配备 64 个深度（Entries）的分支目标缓存（Branch Target Buffer, BTB）。
- 配备 256 个深度（Entries）的分支历史表（Branch History Table, BHT）。
- 配备 2 个深度（Entries）的返回地址堆栈（Return Address Stack, RAS）。
- 配备内存管理单元（Memory Management Unit，MMU）以支持操作系统。
- 配备硬件浮点单元。
- 配备可扩展指令接口（Rocket Custom Coprocessor，RoCC）可供用户扩展协处理器指令。

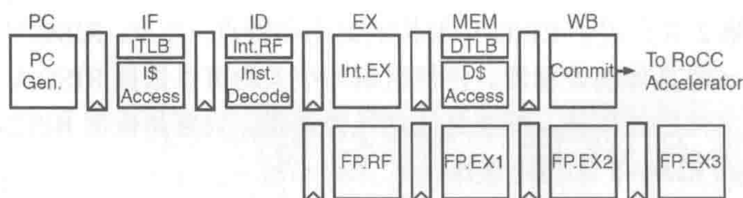


图 3-1 Rocket Core 的流水线结构示意图

据称伯克利使用 Rocket Core 已经成功地进行了高达 11 次的投片，并且在芯片原型上成功地运行了 Linux 操作系统。

Rocket Core 性能和面积等参数非常具有竞争力，伯克利将 Rocket Core 与 ARM Cortex-A5 进行了对比。值得注意的是，Rocket Core 是 64 位的架构，而 Cortex-A5 是 32 位架构，理论上 64 位架构处理器面积和功耗应该远高于 32 位架构的处理器，但是如图 3-2 所示，Rocket Core 与 ARM Cortex-A5 相比性能大幅增加，而面积功耗却更小。

Category	ARM Cortex-A5	RISC-V Rocket
ISA	32-bit ARM v7	64-bit RISC-V v2
Architecture	Single-Issue In-Order	Single-Issue In-Order 5-stage
Performance	1.57 DMIPS/MHz	1.72 DMIPS/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area w/o Caches	0.27 mm^2	0.14 mm^2
Area with 16K Caches	0.53 mm^2	0.39 mm^2
Area Efficiency	2.96 DMIPS/MHz/mm^2	4.41 DMIPS/MHz/mm^2
Frequency	>1GHz	>1GHz
Dynamic Power	<0.08 mW/MHz	0.034 mW/MHz

图 3-2 Rocket Core 与 Cortex-A5 参数对比

由于 Rocket Core 是伯克利推出 RISC-V 架构时，同时推出的开源处理器核，可以说是目前知名的开源 RISC-V Core，有很多的公司与个人均在使用该款处理器进行研究或者开发产品。感兴趣的用户可以在 GitHub 上使用 Rocket-Chip 项目编译出 Rocket Core 自行了解。

Rocket Core 的最大特点是使用 Chisel（Constructing Hardware in an ScalaEmbedded Language）语言进行开发，这是伯克利大学设计的一种开源高层次硬件描述语言，其抽象层次比主流的硬件描述语言 Verilog 要高出许多。Chisel 采用了面向对象，类似于 Java 一样的高层次抽象方式描述电路。这种高层描述语言可以被其工具转换为 Verilog 的 RTL 代码，或者周期精确的 C/C++仿真模型。Chisel 的优点是得益于其面向对象的特性，具有更好的可扩展性与可重用性。正是因为得益于使用了 Chisel 语言，Rocket Core 具备相当程度的可配置性，而如果使用普通的 Verilog 语言开发很难达到这样高度的可配置型和代码的可维护性。

但是每一枚硬币都有其两面性，Chisel 语言虽抽象层次更高，但其转换出来的 Verilog 代码由于是机器生成，其代码类似于电路网表一般，几乎没有可读性，这给像作者这样的用

户造成了很大的困扰。而 Chisel 语言的学习曲线又非常陡峭，难度很大，绝大多数芯片工程师无法看懂，且在繁忙的工作中没有时间来重新学习这么一门非常有难度的新语言。由于硬件工程师无法读懂这种机器生成代码，给后续的 ASIC 流程工作也带来了一些麻烦。因此可以说是喜忧参半，Rocket Core 是一款非常优秀的处理器，但是在相当长一段时间内，作者对于使用 Chisel 语言开发硬件将持非常保守的态度。

3.1.2 BOOM Core （开源）

如上一节所述，BOOM Core 也是伯克利开发的一款开源 RISC-V 处理器核，其也是使用 Chisel 语言开发的，同样需要由伯克利开发的 SoC 生成器 Rocket-Chip 生成。

BOOM 的全称为 Berkeley Out-of-Order Machine，与 Rocket Core 不同的是，BOOM Core 面向更高的性能目标，是一款超标量乱序发射、乱序执行的处理器核。它也配备了高性能的分支预测器，指令 Cache 与数据 Cache 和硬件浮点运算单元，并且还支持多核结构，二级（Level-2）Cache 和多核 Cache 一致性（Coherency），其流水线结构如图 3-3 所示，感兴趣的用户可以在 GitHub 上了解其源代码。

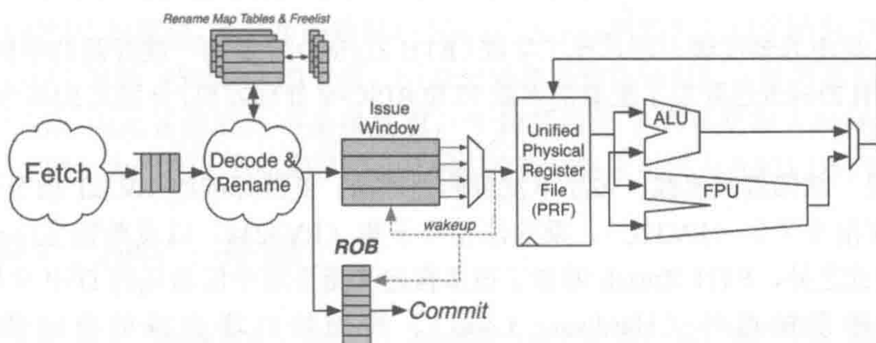


图 3-3 BOOM Core 的流水线结构图

BOOM Core 性能和面积等参数同样非常具有竞争力，伯克利将 BOOM Core 与 ARM Cortex-A9 进行了对比。同样值得注意的是，BOOM Core 是 64 位的架构，而 Cortex-A9 是 32 位架构，理论上 64 位架构处理器面积和功耗应该远高于 32 位架构的处理器，但是如图 3-4 所示，BOOM Core 与 ARM Cortex-A9 相比，性能大幅增加，而面积功耗却更小。

Category	ARM Cortex-A9	RISC-V BOOM-2w
ISA	32-bit ARM v7	64-bit RISC-V v2 (RV64G)
Architecture	2 wide, 3+1 issue Out-of-Order 8-stage	2 wide, 3 issue Out-of-Order 6-stage
Performance	**3.59** CoreMarks/MHz	**4.61** CoreMarks/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area with 32K caches	2.5 mm²	1.00 mm²
Area efficiency	1.4 CoreMarks/MHz/mm²	4.6 CoreMarks/MHz/mm²
Frequency	1.4 GHz	1.5 GHz

图 3-4 BOOM Core 与 Cortex-A9 参数对比

3.1.3 Freedom SoC（开源）

Freedom Everywhere SoC 是由 SiFive 公司推出的一款开源 SoC。SiFive 公司是由伯克利几个主要的 RISC-V 发起人所创办，旨在进行 RISC-V 架构的处理器开发与服务的商业公司。

Freedom Everywhere E310 SoC 是本书将要重点介绍的 SoC，请参阅第 18 章了解此 SoC 的详细信息。

3.1.4 LowRISC SoC（开源）

LowRISC 是一个非营利组织，同时也是由剑桥大学的开发者基于 Rocket Core 而开发的一款开源 SoC 平台名称。LowRISC 组织的口号是希望成为“硬件世界的 Linux（Linux of the hardware world）”，目标是提供高质量、安全、开放的平台，计划将实际量产芯片并提供低成本的开发板，其详情可以从其官网获得。

3.1.5 PULPino Core and SoC（开源）

PULPino 是由苏黎世瑞士联邦理工学院（ETH Zurich）开发的一款开源的单核 MCU SoC 平台，同时 ETH Zurich 还开发了配套的多款 32 位 RISC-V 处理器核，分别是 RI5CY、Zero-riscy 和 Micro-riscy。

RI5CY 是一款四级流水线，按序单发射的处理器，支持标准的 RV32I 指令子集，同时可以配置压缩指令子集（RV32C）、乘除法指令子集（RV32M）以及单精度浮点指令子集（RV32F）。除此之外，ETH Zurich 增加了很多自定义指令用于低功耗的 DSP 应用。这些指令包括硬件协助的循环（Hardware Loop）、带地址自增自减的存储器访问指令（Post-incrementing load and store instructions）、比特操作（Bit-manipulation）、乘累加（MAC）、定点操作（Fixed-point operations）和 SIMD 指令等。

Zero-riscy 是一款二级流水线，按序单发射的处理器，它支持标准的 RV32I 指令子集，同时可以配置压缩指令子集（RV32C）、乘除法指令子集（RV32M），还可以被配置成 16 个通用寄存器版本的 RV32E。该处理器核主要面向的是超低功耗、超小面积的场景。

Micro-riscy 是一款更加小面积的处理器核，它仅支持 16 个通用寄存器版本的 RV32EC 架构，并且没有硬件的乘除法单元，其面积小于 12K 个逻辑门。

RI5CY、Zero-riscy 和 Micro-riscy 的面积对比如图 3-5 所示。

感兴趣的读者可以访问 PULPino 的网站，有丰富的信息与文档可以免费下载。

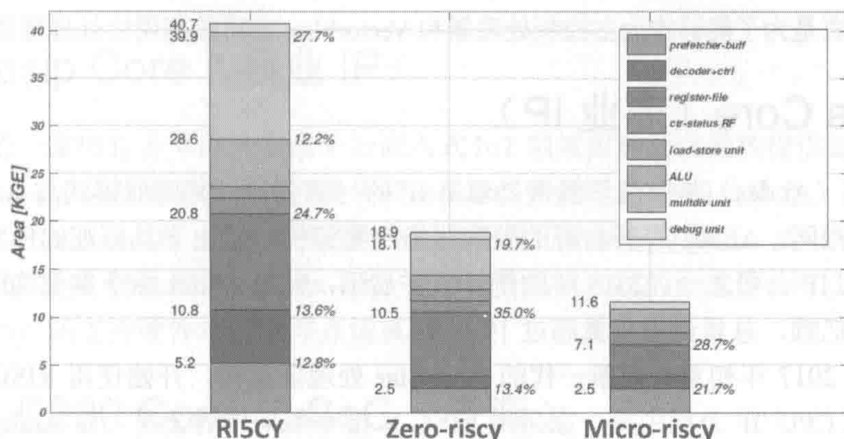


图 3-5 RI5CY、Zero-riscy 和 Micro-riscy 的面积对比

3.1.6 PicoRV32 Core（开源）

PicoRV32 是一款由著名的 IC 设计师 Clifford Wolf 开发并开源的一款 RISC-V 处理器核。Clifford Wolf 由于撰写多篇知名的数字 IC 设计论文而被人所熟知。这款 PicoRV32 的重点在于追求面积和频率的优化，其公布的数据在 Xilinx7-Series FPGA 上的开销为 750-2000 LUTs，并且能够综合到 250～450MHz 的主频。但是此处理器核明确说明它是为面积做优化，而非为性能做优化，因此其性能并不是很理想，平均每条指令的周期数（Average cycles Per Instruction，CPI）大约在 4 个周期，Dhrystone 的跑分结果也仅为 0.521 DMIPS/MHz。

3.1.7 SCR1 Core（开源）

SCR1 是一款由 Syntacore 公司使用 System Verilog 语言设计编写的一款极低功耗开源 RISC-V 处理器核。Syntacore 是一家俄罗斯公司，专注于为客户定制开发和授权具有高能效比的可综合可编程处理器核。Syntacore 公司基于 RISC-V 架构开发了多款针对 MCU 级别的处理器核，被称为 SCRx 系列，并将其中的最简单款 SCR1 开源。

SCR1 具有可配置的特性，可以配置为 RV32I/EMC 指令子集的组合，最小配置 RV32EC 的面积开销为 12K 个逻辑门，最高的 RV32IMC 配置面积开销为 28K 左右个逻辑门。SCR1 仅支持机器模式，同时还配备了可选的中断控制器与调试（Debugger）模块。

3.1.8 ORCA Core（开源）

ORCA 是一款由 Vectorblox 公司使用 VHDL 语言设计编写的面向 FPGA 的开源 RISC-V 处理器核，可以配置成为 RV32I 或者 RV32M。虽然 ORCA 也可以作为一种单独的处理器核使用，

但是其诞生初衷是为了能够作为主控制处理器和 Vectorblox 公司的商用协处理器适配使用。

3.1.9 Andes Core（商业 IP）

晶心科技（Andes）是专注于提供处理器 IP 的一家公司，其商业模式与 ARM 这样的处理器 IP 公司相同。Andes 有其自有的处理器指令集架构，且由于其可观的出货量，一直是商用主流 CPU IP 公司之一。2016 年的统计数字显示，采用 Andes 指令集架构的系统芯片出货量超过 4.3 亿颗，总累计出货量超过 19 亿颗。

Andes 于 2017 年初发布最新一代的 AndeStar 处理器架构，开始使用 RISC-V 指令集，成为商用主流 CPU IP 公司中第一家采用 RISC-V 指令集架构的公司。在 Andes 的介绍中表示，AndeStarV5 架构不但将 RISC-V 兼容性完全纳入，同时也包含多项 Andes 独创的通用便利功能及应用强化单元。

作为一个有多年历史的商用主流 CPU IP 公司，其在下一代的主要架构中开始全面采用 RISC-V 架构，具有非常重要的代表意义。

3.1.10 Microsemi Core（商业 IP）

Microsemi 公司的 FPGA 由于其高可靠性被人所熟知，被广泛应用于对可靠性要求苛刻的场景。Microsemi 也是最早支持并使用 RISC-V 处理器的公司之一。推出了业界首个基于 RISC-V 内核的 FPGA 系列产品，即 IGLOO2 FPGA、SmartFusion2 SoC FPGA 或 RTG4 FPGA。Microsemi 称完全开源的 RV32IM RISC-V 内核采用开放式指令集架构具备全面可移植性，而且由于开发人员可以查看 RISC-V 的所有源码，因此安全性更高，再加上 Microsemi 产品一向出色的功耗与可靠性指标，非常适合现在嵌入式应用对于平台架构的要求。

在 Microsemi 的介绍中表示，集成 RISC-V 内核的 FPGA 的特色主要是在开放性、可移植性和设计灵活性方面表现得更好。RISC-V 内核特别适合高效的设计实现，开发人员可以根据应用需求灵活裁剪。如今 RISC-V 指令集已经固定，全部 RISC-V 指令不超过 50 个，因此 RISC-V 内核面积更小，从而使得整体芯片成本更低，内核越小，相应的功耗也就越低。相比 MCU 或集成商用处理器核的 FPGA，RISC-V 内核的 FPGA 最大的优势之一就是可移植性。采用 FPGA 来开发新应用能够快速上市，如果该应用成熟以后有足够多的量，那么可以将 FPGA 改为专用芯片来降低成本。采用 ARM 核就没这么方便了，不支付一笔价格不菲的工程费用和专利费是无法完成的。RISC-V 的开放性也是一大优点，采用 ARM 等封闭式架构内核的平台，开发人员看不到源代码，所以无法了解门级电路设计细节。但 RISC-V 的用户可以查看内核的所有细节，可以检查每一行代码以确定系统的安全，甚至根据需要定制自己的安全模块。

3.1.11 Codasip Core（商业 IP）

Codasip 是一家拥有多年经验专注于为嵌入式 IoT 领域定制处理器核提供处理器 IP 和服务的公司，Codasip 是最早正式设计并提供商用 RISC-V 处理器 IP 的公司之一。目前该公司提供 Codix-BK Processor IP，支持多种的指令子集配置可定制指令接口。其中 Codix-BK3 是一款三级流水线的 32 位处理器；Codix-BK5 是一款五级流水线的处理器，可以配置为 32 位或者 64 位，同时还支持硬件单精度浮点运算器。

3.1.12 蜂鸟 E200 Core 与 SoC（开源）

在上述的各小节中，我们了解了诸多的开源与商用 RISC-V 处理器核，既有特点，也有缺点。

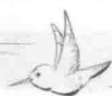
本书将介绍一款独特的开源 RISC-V 处理器核和配套 SoC——蜂鸟 E200 系列处理器核与 SoC，它有效地克服了当前开源处理器的诸多缺点。请参见第 4 章了解更多详情。

3.2 总结

得益于 RISC-V 开放免费的特点，全球在极短的时间内便涌现出了众多版本的 RISC-V 处理器核，有道是“乱花渐欲迷人眼，浅草才能没马蹄”。各开源处理器都有什么优缺点？蜂鸟 E200 开源处理器系列又有何过人之处呢？欲闻其详，且看下一章详细分解。

第4章　开源 RISC-V——蜂鸟 E200 系列超低功耗 Core 与 SoC

小个子有大力量





蜂鸟 E200 系列处理器由作者所在的公司开发，是一款开源 RISC-V 处理器。蜂鸟是世界上最小的鸟类，其体积虽小，却有着极高的速度与敏锐度，可以说是“能效比”最高的鸟类。E200 系列以蜂鸟命名便寓意于此，旨在将其打造成为一款世界上最高能效比的 RISC 处理器。

注意：本章对蜂鸟 E200 处理器的介绍将使用许多处理器的关键特性参数或名称，对于完全不了解 CPU 的初学者而言可能难以理解，请参见本书第二部分与第三部分的各章节进行系统学习后再行理解本章。

4.1 与众不同的蜂鸟 E200 处理器

在第 3 章中介绍了诸多的开源与商用 RISC-V 处理器核，对于商业公司提供的付费 IP 本文不加以评述，但是对于众多开源实现加以分析，可以发现如下现象。

- 目前开源的 RISC-V 实现主要以国外为主，难以取得本土开发人员的交流和支持。
- 面向 IoT 领域的高性能且超低功耗的开源 RISC-V 处理器，可以选择的并不多，能效表现也难以对目前 IoT 领域的主流商用 ARM Cortex-M 系列处理器（2 级或者 3 级流水线实现）形成有效的替代。
- 绝大多数的开源处理器仅提供处理器核的实现，没有提供配套 SoC 和软件示例，用户若要将其使用起来，且移植完整软件需要额外付出不小的努力。
- 大多数的开源实现或来自个人爱好者或来自高校。其开发语言或使用 VHDL，或使用 System Verilog。来自产业界工程团队，且使用最稳健的 Verilog RTL 实现的开源 RISC-V 处理器尚不多见。
- 有些开源 RISC-V 处理器使用高级的 Chisel 语言转换生成 Verilog RTL 代码，造成代码可读性很差，给业界只熟悉 Verilog 的芯片工程师使用造成了困难。
- 绝大多数开源处理器仅提供处理器核的实现，但是并没有提供调试方案的实现，很少有开源处理器能够支持完整的 GDB 交互调试功能。
- 绝大多数开源处理器均文档比较匮乏。

以上是许多国内用户接触 RISC-V 并选择超低功耗开源处理器核时遇到的困难。蜂鸟 E200 系列处理器可有效解决以上这些问题，与其他的 RISC-V 开源处理器实现相比，它具有如下显著特点。

- 蜂鸟 E200 系列是一个开源的 RISC-V 处理器。蜂鸟 E200 系列由中国大陆研发团队开发，用户能够轻松与开发人员取得交流和支持。
- 蜂鸟 E200 处理器研发团队拥有在国际一流公司多年开发处理器的经验，使用稳健的

Verilog 2001 语法编写的可综合 RTL 代码，以工业级标准进行开发。

- 蜂鸟 E200 的代码为人工编写，添加丰富的注释且可读性强，非常易于理解。
- 蜂鸟 E200 专为 IoT 领域量身定做，其具有 2 级流水线深度，功耗和性能指标均优于目前主流商用的 ARM Cortex-M 系列处理器，且免费开源，能够在 IoT 领域完美替代 ARM Cortex-M 处理器。
- 蜂鸟 E200 不仅提供处理器核的实现，还提供完整的配套 SoC、详细的 FPGA 原型平台搭建步骤，详细的软件运行实例。用户可以按照步骤重现出整套 SoC 系统，轻松将 E200 处理器核应用到具体产品中。
- 蜂鸟 E200 不仅提供处理器核的实现、SoC 实现、FPGA 平台和软件示例，还实现了完整的调试方案，具备完整的 GDB 交互调试功能。蜂鸟 E200 是从硬件到软件，从模块到 SoC，从运行到调试的一套完整解决方案。
- 蜂鸟 E200 系列提供丰富的文档和实例，本书亦专门对其源代码进行完整的剖析。

蜂鸟 E200 的开源口号是：让免费的蜂鸟 E200 成为中国的下一个 8051，为中国 IoT 领域的发展助力提速。感兴趣的读者可以在互联网上搜索作者曾发表过的文章《进入 32 位时代，谁能成为下一个 8051》。

蜂鸟 E200 开源项目的源代码托管于著名开源网站 GitHub。GitHub 是一个世界著名的免费的项目托管网站，任何用户无须注册即可从网站上下载源代码，众多的开源项目均将源代码托管于此。E200 项目网址请在 GitHub 中搜索“e200_opensource”。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

4.2 蜂鸟 E200 简介——蜂鸟虽小，五脏俱全

蜂鸟 E200 主要面向极低功耗与极小面积的场景，非常适合于替代传统的 8051 内核或者 Cortex-M 系列内核应用于 IoT 或其他低功耗场景。同时，蜂鸟 E200 作为结构精简的处理器核，可谓“蜂鸟虽小，五脏俱全”，源代码全部开源公开，文档详实，非常适合作为大中专院校师生学习 RISC-V 处理器设计（使用 Verilog 语言）的教学或自学案例。

蜂鸟 E200 系列处理器核的特性简介如下。

- E200 系列处理器核采用两级流水线结构，通过一流的处理器架构设计。该 CPU 核的功耗与面积均优于同级 ARM Cortex-M 核，实现业界最高的能效比与最低的成本。
- E200 系列处理器核能够运行 RISC-V 指令集，支持 RV32I/E/A/M/C/F/D 等指令子集的配置组合，支持机器模式（Machine Mode Only）。
- E200 系列处理器核提供标准的 JTAG 调试接口以及成熟的软件调试工具。
- E200 系列处理器核提供成熟的 GCC 编译工具链。

- E200 系列处理器核配套 SoC 提供紧耦合系统 IP 模块，包括中断控制器、计时器，UART、QSPI 和 PWM 等，即时能用（Ready-to-Use）的 SoC 平台与 FPGA 原型系统。

蜂鸟 E200 系列处理器的系统示意图如图 4-1 所示，其提供丰富的存储和接口如下。

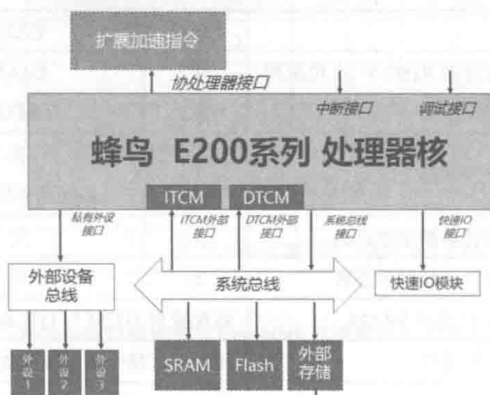


图 4-1 蜂鸟 E200 处理器系统示意图

- 私有的 ITCM（指令紧耦合存储）与 DTCM（数据紧耦合存储），实现指令与数据的分离存储同时提高性能。
- 中断接口用于与 SoC 级别的中断控制器连接。
- 调试接口用于与 SoC 级别的 JTAG 调试器连接。
- 系统总线接口，用于访存指令或者数据。可以将系统主总线接到此接口上，E200 可以通过该总线访问总线上挂载的片上或者片外存储模块。
- 紧耦合的私有外设接口，用于访存数据。可以将系统中的私有外设直接接到此接口上，使得 E200 无须经过与数据和指令共享的总线便可访问这些外设。
- 紧耦合的快速 IO 接口，用于访存数据。可以将系统中的快速 IO 模块直接接到此接口上，使得 E200 无须经过与数据和指令共享的总线便可访问这些模块。
- 所有的 ITCM、DTCM、系统总线接口、私有外设接口以及快速 IO 接口均可以配置地址区间。

4.3 蜂鸟 E200 型号系列

蜂鸟 E200 是一个处理器系列，包含了多款不同的具体处理器型号。所有的 E200 系列处理器核均支持协处理器接口，可用于自定义扩展指令，不同型号的具体差别如表 4-1 所示。

- E201 核是面积最小的核，可以配置为 RV32IC 或者 RV32EC 架构，不支持其他的扩展指令子集。
- E203 核可以配置为 RV32IMAC 或者 RV32EMAC 架构，使用面积优化的多周期硬件乘除法单元。
- E205 核为 RV32IMAC 架构，使用单周期的硬件乘法单元和多周期的硬件除法单元。
- E205f 核为 RV32IMAFC 架构，在基本的 E205 基础上加入了单精度浮点运算单元。
- E205fd 核为 RV32IMAFDC 架构，在基本的 E205 基础上加入了单精度和双精度浮点运算单元。

注意：目前蜂鸟 E200 系列中开源的处理器型号为 E203。

表 4-1　E200 处理器型号特性介绍

	E201	E203	E205	E205f	E205fd
支持的 RISC-V 32 位架构指令子集	IC EC	IMAC EMAC	IMAC	IMAFC	IMAFDC
硬件乘除法单元	无	有 （多周期乘除法器）	有 （单周期乘法器，多周期除法器）		
硬件单精度浮点器	无	无	无	有	有
硬件双精度浮点器	无	无	无	无	有
ECC 保护 SRAM	可配置对 ITCM 与 DTCM 进行 ECC 保护				
可扩展性	可以进行指令集扩展（支持协处理器接口）				
基本外设	提供 SPI、UART、PWM、GPIO、Timer、中断控制器等基本外设				

4.4 蜂鸟 E200 性能指标

蜂鸟 E200 处理器核的功耗与面积以及性能参数非常有竞争力，如表 4-2 所示。

- 蜂鸟 E203 核的功耗面积和性能不逊色于 ARM 的 Cortex-M0+处理器核（M0+ 是 ARM 最小面积的处理器核）。
- 蜂鸟 E205 核的功耗面积和性能不逊色于 Cortex-M3 处理器核。
- E205fd 提供了目前仅在 Cortex-M7 中才具备的双精度浮点特性（Cortex-M7 是面积较大的处理器核，相比蜂鸟 E205fd 而言，功耗会更大）。

表 4-2　“蜂鸟 E200 系列核”与“ARM Cortex-M 系列核”对比表

	ARM Cortex-M0	ARM Cortex-M0+	ARM Cortex-M3	蜂鸟 E201	蜂鸟 E203	蜂鸟 E205
Dhrystone (DMIPS/MHz)	0.84 （正式数据） 1.21 （经选项最大优化后的数据）	0.94 （正式数据） 1.31 （经选项最大优化后的数据）	1.25	1.171	1.23	1.355
CoreMark (CoreMark/MHz)	2.33	2.42	3.32	1.352	2.14	3.327
最小配置逻辑门数 (K Gates)	12K	12K	36K	10K	12K	20K
频率	不详			180nm SMIC 工艺下 50～100MHz		

续表

	ARM Cortex-M0	ARM Cortex-M0+	ARM Cortex-M3	蜂鸟 E201	蜂鸟 E203	蜂鸟 E205
流水线深度	3	2	3	2	2	2
乘法器	有	有	有	无	有	有
除法器	无	无	有	无	有	有
ITCM（指令紧耦合存储）DTCM（数据紧耦合存储）	不提供内嵌的 ITCM 与 DTCM			提供内嵌的 ITCM 与 DTCM		
可扩展性	不支持指令集扩展			支持协处理器接口进行指令扩展		

注意：

- Cortex-M0+的乘法器可以配置成单周期乘法器或多周期迭代乘法器。Dhrystone 性能数据与 CoreMark 性能数据是采用单周期乘法还是多周期乘法器的信息不详
- 本表格中有关 ARM Cortex-M 系列处理器核的性能数据来自于本书撰写之时收集的公开信息和非官方数据，本书对其正确性不做保证，请读者以最新 ARM 官方数据为准

4.5 蜂鸟 E200 配套 SoC

很多开源的处理器核仅提供其实现，为了能够完整使用，用户需要花费不少精力来构建完整的 SoC 平台、FPGA 平台。很多开源的处理器核也不提供对于调试器（Debugger）的支持。为了方便用户快速地上手使用，蜂鸟 E200 不仅开源了自主设计的 Core，还开源如下配套组件。

- 完整的 SoC 平台，如图 4-2 和表 4-3 所示。有关 SoC 的详情请参见第 18.2 节。

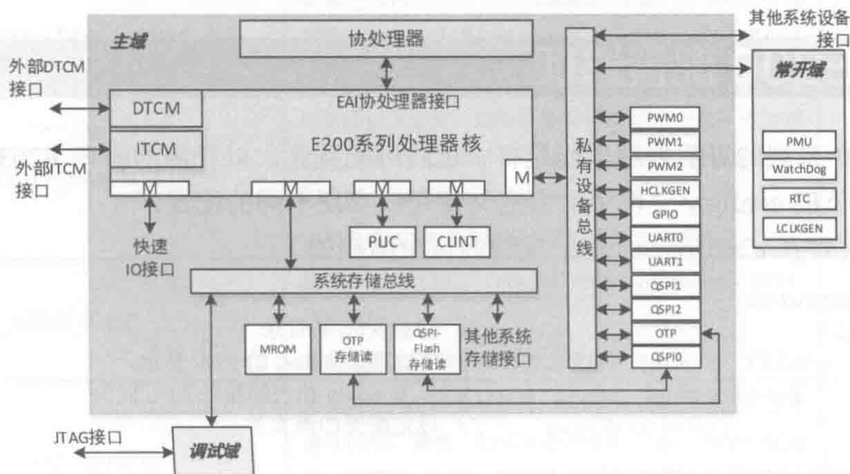


图 4-2　E200 系列处理器配套 SoC 结构示意图

表 4-3　　E200 系列处理器配套 SoC 特性

	特　性	描　述
CPU	使用 Windows/Linux GCC 工具链开发	—
	基于 E200 系列处理器核	—
	使用标准 JTAG 调试接口，支持 GDB 交互式软件调试能力	—
	支持中断控制器	—
存储	片上 ITCM-SRAM（指令）	可配置大小
	片上 DTCM-SRAM（数据）	可配置大小
	可通过 QSPI 等接口外接其他片外存储	片外 Flash
外设	提供 PWM	3 组
	提供 SPI、QSPI	3 组
	提供 GPIO	32 个 pin 脚
	提供 UART	1 组
	提供 WatchDog	1 组
	提供 RTC（Real Time Counter）	1 组
	提供计时器（Timer）	1 组

- 提供软件开发环境，如第 19.2 节所述。
- 提供交互式硬件调试工具（GDB）的支持，如第 19.4 节所述。

用户可按照第 17、18 章所描述的步骤便可快速搭建完整的 SoC 仿真平台和 FPGA 原型平台，按照第 19 章所描述的步骤便可快速运行软件示例。可以说，蜂鸟 E200 开源的不仅是一个处理器核，而且是一个完整 MCU 软硬件实现。

4.6 蜂鸟 E200 配置选项

蜂鸟 E200 系列的每款处理器均具有一定的可配置性。以开源的蜂鸟 E203 核为例，通过修改其目录下的 config.v 文件中的宏定义便可以实现不同的配置。

config.v 文件在 e200_opensource 项目的目录结构如下。

```
e200_opensource
 |----rtl                             // 存放 RTL 的目录
      |----e203                       // E203 核和 SoC 的 RTL 目录
           |----core                  // 存放 core 相关模块的 RTL 代码
                |----config.v         // 设定配置的源文件
```

config.v 中的具体的配置选项宏定义如表 4-4 所示。

表 4-4　　E203 处理器核配置选项

宏	描　述	推荐默认值
E203_CFG_DEBUG_HAS_JTAG	如果添加了此宏，则配置使用 JTAG 调试接口 请参见第 14 章了解有关调试器信息	使用
E203_CFG_ADDR_SIZE_IS_16 E203_CFG_ADDR_SIZE_IS_24 E203_CFG_ADDR_SIZE_IS_32	此 3 个宏选择其中一个，用于配置处理器的总线地址宽度为 16 位、24 位和 32 位	32 位
E203_CFG_SUPPORT_MCYCLE_MINSTRET	如果添加了此宏，则配置使用 MCYCLE 和 MINSTRET 这两个 64 位的计数器 请参见附录 B2.17 和 B2.18 节了解有关 MCYCLE 和 MINSTRET 寄存器信息	使用
E203_CFG_REGNUM_IS_32 E203_CFG_REGNUM_IS_16	此两个宏选择其中一个，用于配置整数通用寄存器组使用 32 个通用寄存器（RV32I）还是 16 个通用寄存器（RV32E） 请参见附录 A4.1 节了解有关通用寄存器组的信息	32 个
E203_CFG_HAS_ITCM	如果添加了此宏，则配置使用 ITCM 请参见第 11.4.4 节了解有关 ITCM 的信息	使用
E203_CFG_ITCM_ADDR_BASE	配置 ITCM 的基地址	0x8000_0000
E203_CFG_ITCM_ADDR_WIDTH	配置 ITCM 的大小，使用地址总线宽度作为其大小的衡量。譬如，假设 ITCM 的大小为 1KByte，则此宏定义值为 10	16 （64KB）
E203_CFG_HAS_DTCM	如果添加了此宏，则配置使用 DTCM 请参见第 11.4.4 节了解有关 DTCM 的信息	使用
E203_CFG_DTCM_ADDR_BASE	配置 DTCM 的基地址	0x9000_0000
E203_CFG_DTCM_ADDR_WIDTH	配置 DTCM 的大小，使用地址总线宽度作为其大小的衡量，例如假设 DTCM 的大小为 1KByte，则此宏定义值为 10	16 （64KB）
E203_CFG_REGFILE_LATCH_BASED	如果添加了此宏，则配置使用锁存器（Latch）作为通用寄存器组（Regfile）的基本单元；如果没有添加此宏，则使用 D 触发器作为基本单元 请参见第 8.3.4 节了解有关寄存器组实现的信息	不使用锁存器
E203_CFG_PPI_ADDR_BASE	配置私有外设接口（Private Peripheral Interface，PPI）的基地址 请参见第 12.4 节了解有关总线接口的信息	0x1000_0000
E203_CFG_PPI_BASE_REGION	配置 PPI 接口的地址区间，通过指定高位的区间来界定地址区间。譬如，如果该宏定义为 31:28，基地址定义为 0x1000_0000，则表示 PPI 的地址区间为 0x1000_0000～0x1FFF_FFFF	31:28

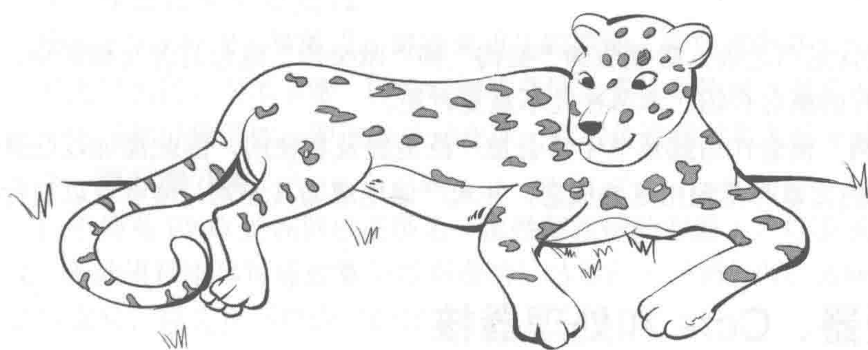
续表

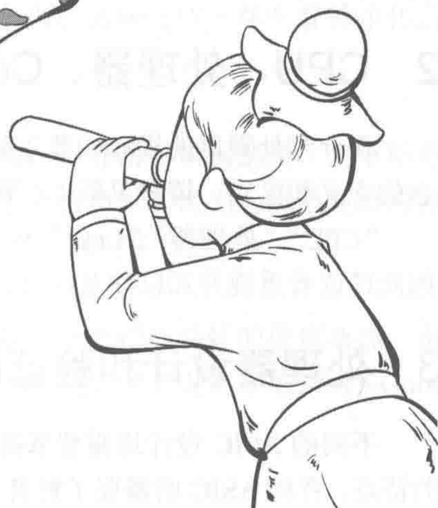
宏	描　述	推荐默认值
E203_CFG_FIO_ADDR_BASE	配置快速 FIO 接口（Fast IO Interface，FIO）的基地址 请参见第 12.4 节了解有关总线接口的信息	0xf000_0000
E203_CFG_FIO_BASE_REGION	配置 FIO 接口的地址区间，通过指定高位的区间来界定地址区间。例如如果该宏定义为 31:28，基地址定义为 0xf000_0000，则表示 FIO 的地址区间为 0xf000_0000～0xfFFF_FFFF	31:28
E203_CFG_CLINT_ADDR_BASE	配置 CLINT 接口的基地址 请参见第 13.5.5 节了解有关 CLINT 的信息	0x0200_0000
E203_CFG_CLINT_BASE_REGION	配置 CLINT 接口的地址区间，通过指定高位的区间来界定地址区间。例如如果该宏定义为 31:16，基地址定义为 0x0200_0000，则表示 PLIC 的地址区间为 0x0200_0000～0x0200_FFFF	31:16
E203_CFG_PLIC_ADDR_BASE	配置 PLIC 接口的基地址 请参见第 13.5.6 节了解有关 PLIC 的信息	0x0C00_0000
E203_CFG_PLIC_BASE_REGION	配置 PLIC 接口的地址区间，通过指定高位的区间来界定地址区间。例如如果该宏定义为 31:24，基地址定义为 0x0C00_0000，则表示 PLIC 的地址区间为 0x0C00_0000～0x0CFF_FFFF	31:24
E203_CFG_HAS_ECC	如果添加了此宏，则配置使用 ECC 对 ITCM 和 DTCM 的 SRAM 进行保护 注意：在 GitHub 上，此选项的功能并未开源，因此相关代码并不具备，即便添加了配置宏也不起作用	不使用 ECC
E203_CFG_HAS_EAI	如果添加了此宏，则配置使用协处理器接口 注意：在 GitHub 上，此选项的功能并未开源，因此相关代码并不具备，即便添加了配置宏也不起作用	无协处理器接口
E203_CFG_SUPPORT_SHARE_MULDIV	如果添加了此宏，则配置使用面积优化的多周期乘除法单元	使用多周期乘除法
E203_CFG_SUPPORT_AMO	如果添加了此宏，则支持 RISC-V 的“A”扩展指令子集 请参见附录 A14.5 节了解 RV32A 指令子集的信息	支持 RISC-V 的“A”扩展指令子集

第二部分

手把手教你使用 Verilog 设计 CPU

第 5 章　先见森林，后观树木
——蜂鸟 E200 设计总览和顶层介绍





管中窥豹只见一斑，
这不是正确的打开方式。
应该先看全局，
再观局部。

在学习或者讲解某个技术要点时，作者比较推崇的方法是“先见森林，后观树木”，即先从宏观着手了解全局，然后才切入微观细节。

本书的第二部分中的各章节将以蜂鸟 E200 为具体实例介绍如何设计一款 RISC-V CPU。作为本部分的开篇之章，本章将先从宏观的角度着手，介绍若干处理器设计的总览要诀、蜂鸟 E200 处理器核的总体设计思想和顶层接口。

通过本章的学习，将帮助读者整体认识蜂鸟 E200 处理器的设计要诀，为后续各章针对不同部分的展开详述而奠定基础。

注意：蜂鸟 E200 开源项目的所有源代码均托管于著名开源网站 GitHub，本书所有章节中将广泛地以“e200_opensource 项目”为简称代表 GitHub 网站上蜂鸟 E200 的项目路径（请在 GitHub 中搜索“e200_opensource”）。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

5.1 处理器硬件设计概述

5.1.1 架构和微架构

在介绍处理器的设计细节之前，必须明确“架构”和“微架构”概念的含义和区别。请参见第 1.1.1 节对其进行的概念介绍，本节在此不重复讨论。

“架构”和“微架构”概念在后续章节中将会被广泛地提及和使用，因此需加以注意区分。当然，网络上很多的文章时常混用这些概念，并未严谨地遵循其差别，读者可以自行体会上下文予以甄别。

5.1.2 CPU、处理器、Core 和处理器核

在介绍处理器的设计细节之前，还须明确“CPU”“处理器”“Core”和“处理器核”概念的含义和区别。请参见第 1.1 节中的概念介绍，本节在此不重复讨论。

“CPU”“处理器”“Core”和“处理器核”概念在后续章节中将会被广泛地提及和使用，因此请读者重视并加以注意区分。

5.1.3 处理器设计和验证的特点

不同的 ASIC 设计均需要掌握不同的背景知识，例如通信 ASIC 需要我们了解通信算法的特点，音频 ASIC 则需要了解音频算法的特点。而处理器设计作为一种特殊的 ASIC 设计，

也需我们了解某些方面的背景知识和特点，归纳如下。

- 熟悉汇编语言及其执行过程。
- 了解软件如何经编译、汇编、链接最后成为处理器可执行的二进制码的过程。
- 了解计算机体系结构的知识。
- 处理器对时序和面积的要求一般会非常严格，需不断反复地优化时序和面积，因此对电路和逻辑设计的理解需要比较深刻。

处理器的验证也有其特点，通常需要从3个不同层面对处理器进行验证。

- 需使用传统的模块级验证手段（例如UVM等）对处理器的子模块进行验证。
- 需使用人工编写或者随机生成的汇编语言测试用例在处理器上运行进行验证。
- 需使用高等语言（譬如C、C++）编写的测试用例在处理器上运行进行验证。

综上所述，处理器的设计和验证是一个软硬件联合的过程，牵涉的方面比较多，工作量比较大。

5.2 蜂鸟E200处理器核设计哲学

1. 模块化和可重用性

蜂鸟E200处理器核的设计遵循模块化的原则，将处理器划分为几个主体模块单元，每个单元之间的接口简单清晰，而将盘根错节的关系尽量控制在单元内部。在划分模块单元时还充分考虑到可重用性，即这些单元在下一代的处理器核微架构中还能够被继续使用。

2. 面积最小化

由于蜂鸟E200处理器核在满足一定性能指标的前提下，以追求低功耗、小面积为第一要义，因此设计中尽可能地复用数据通路以节省面积开销。当在某些细节上存在着时序和面积的冲突时，应选择面积优先的策略。

3. 结构简单化

蜂鸟E200处理器核在设计哲学上与RISC-V架构一致，即遵循简单就是美，简单即可靠的策略。在微架构的制定上防止陷入繁复的陷阱，在有选择的情形下优先选用最简单的方案，只有在最关键的场景才使用复杂的设计方案，即所谓“好钢用在刀刃上”。

4. 性能不追求极端

处理器对于性能的追求往往是严格的。由于蜂鸟E200是一款超低功耗的处理器核，虽然也追求性能的最大化，但须以面积最小化和结构简单化为前提，性能提倡够用即可（Good Enough）的哲学。

5.3 蜂鸟 E200 处理器核 RTL 代码风格介绍

蜂鸟 E200 处理器核采用一套统一的 Verilog RTL 编码风格（Coding Style），来自于严谨的工业级开发标准，其要点如下。

- 使用标准 DFF 模块例化生成寄存器。
- 推荐使用 Verilog 中的 assign 语法替代 if-else 和 case 语法进行代码编写。
- 其他若干注意事项。

下面分别予以详述。

5.3.1 使用标准 DFF 模块例化生成寄存器

寄存器是数字同步电路中最基本的单元。使用 Verilog 进行数字电路设计时，最常见的方式是使用 always 块语法生成寄存器。蜂鸟 E200 处理器核推荐如下原则，本原则来自于严谨的工业级开发标准，其要点如下。

（1）对于寄存器避免直接使用 always 块编写，而是应该采用模块化的标准 DFF 模块进行例化。示例如下所示，一个名为 flg_dfflr 的寄存器，除了时钟（clk）和复位信号（rst_n）之外，还带有使能信号 flg_ena 和输入（flg_nxt）/输出信号（flg_r）。

```
   wire flg_r;
   wire flg_nxt = ~flg_r;
   wire flg_ena = (ptr_r == ('E203_OITF_DEPTH-1)) & ptr_ena;

 //此处使用例化 sirv_gnrl_dfflr 的方式实现寄存器，而不是使用显示的 always 块
sirv_gnrl_dfflr #(1) flg_dfflrs(flg_ena, flg_nxt, flg_r, clk, rst_n);
```

（2）使用标准 DFF 模块例化的好处包括以下内容。

- 便于全局替换寄存器类型。
- 便于在寄存器中全局插入延迟。
- 明确的 load-enable 使能信号（如下例的 flg_ena）方便综合工具自动插入寄存器级别的门控时钟以降低动态功耗（参见第 15.1.5 节了解更多此低功耗设计的信息）。
- 便于规避 Verilog 语法 if-else 不能传播不定态的问题，有关此内容可以参考第 5.3.2 节。

（3）标准 DFF 模块是一系列不同的模块，相关源代码在 e200_opensource 目录的结构如下。

```
e200_opensource
    |----rtl                                   // 存放 RTL 的目录
         |----e203                             // E203 核和 SoC 的 RTL 目录
              |----general                     // 存放一些通用模块的 RTL 代码
                   |----sirv_gnrl_dffs.v       // 该文件中编写了一系列不同的 DFF 模块，
```

列举如下：

```
sirv_gnrl_dfflrs   //带有 load-enable 使能，带有异步 reset，复位默认值为 1 的寄存器
sirv_gnrl_dfflr    //带有 load-enable 使能，带有异步 reset，复位默认值为 0 的寄存器
sirv_gnrl_dffl     //带有 load-enable 使能，不带有 reset 的寄存器
sirv_gnrl_dffrs    //不带有 load-enable 使能，带有异步 reset，复位默认值为 1 的寄存器
sirv_gnrl_dffr     //不带有 load-enable 使能，带有异步 reset，复位默认值为 0 的寄存器
sirv_gnrl_ltch     //Latch 锁存器模块
```

（4）标准 DFF 模块内部则使用 Verilog 语法的 always 块进行编写，以 dfflr 为例，如下所示。由于 Verilog if-else 语法不能传播不定态，对处于 if 条件中的 lden 信号为不定态的非法情况使用断言（assertion）进行捕捉。

```
// 标准 DFF 模块，以 sirv_gnrl_dfflr 为例，代码片段如下

module sirv_gnrl_dfflr # (
  parameter DW = 32
) (

  input               lden,
  input      [DW-1:0] dnxt,
  output     [DW-1:0] qout,

  input               clk,
  input               rst_n
);

reg [DW-1:0] qout_r;

    //使用 always 块编写寄存器逻辑
always @(posedge clk or negedge rst_n)
begin : DFFLR_PROC
  if (rst_n == 1'b0)
    qout_r <= {DW{1'b0}};
  else if (lden == 1'b1)
    qout_r <= dnxt;
end

assign qout = qout_r;

    //使用 assertion 捕捉 lden 信号的不定态
'ifndef FPGA_SOURCE//{
'ifndef SYNTHESIS//{
sirv_gnrl_xchecker # ( //该模块内部是个 SystemVerilog 编写的断言，如下文详解
  .DW(1)
) u_sirv_gnrl_xchecker(
  .i_dat(lden),
  .clk  (clk)
);
'endif//}
'endif//}
```

```
endmodule


// sirv_gnrl_xchecker 模块代码片段

// 此模块专门捕捉不定态，一旦输入的 i_dat 出现不定态，则会报错并终止仿真。
module sirv_gnrl_xchecker # (
  parameter DW = 32
) (
  input  [DW-1:0] i_dat,
  input clk
);


CHECK_THE_X_VALUE:
  assert property (@(posedge clk)
                    ((^(i_dat)) !== 1'bx)
                  )
  else $fatal ("\n Error: Oops, detected a X value!!! This should never hap
pen. \n");

endmodule
```

5.3.2 推荐使用 assign 语法替代 if-else 和 case 语法

Verilog 中的 if-else 和 case 语法存在两大缺点。

- 不能传播不定态。
- 会产生优先级的选择电路而非并行选择电路，从而不利于时序和面积。

为了规避这两大缺点，蜂鸟 E200 处理器核推荐使用 assign 语法进行代码编写，本原则来自于严谨的工业级开发标准，详细解释如下。

```
问题一：

Verilog 的 if-else 不能传播不定态，以如下代码片段为例。假设 a 的值为 X 不定态，按照 Verilog
语法会将其等效于 a == 0，从而让 out 输出值等于 in2 最终没有将 X 不定态传播出去。这种情况可能会在
仿真阶段掩盖某些致命的 bug，造成芯片功能错误。
    if(a)
        out = in1;
    else
        out = in2;
//而使用功能等效的 assign 语法，如下所示，假设 a 的值为 X 不定态，按照 Verilog 语法，则会将 X
不定态传播出去，从而让 out 输出值也等于 X。通过 X 不定态的传播，可以在仿真阶段将 bug 彻底暴露出来
    assign out = a ? in1 : in2;
//虽然现在有的 EDA 工具提供专有选项（例如 Synopsys VCS 提供 xprop 选项）可以将 Verilog 原
始语法中定义的“不传播不定态”的情形强行传播出来，但是一方面不是所有的 EDA 工具均支持此功能；另
一方面在操作中此选项也时常被忽视，从而造成疏漏
```

问题二：

Verilog 的 Case 语法也不能传播不定态，与问题一中的 if-else 同理。而使用等效的 assign 语法即可规避此缺陷。

问题三：

Verilog 的 if-else 语法会被综合成为优先级选择的电路，面积和时序均不够优化，如下所示：

```
if(sel1)
    out = in1[3:0];
else if (sel2)
    out = in2[3:0];
else if (sel3)
    out = in3[3:0];
else
    out = 4'b0;
```

如果此处确实是希望生成一种优先级选择的逻辑，则推荐使用 assign 语法等效地编写成如下形式，以规避 X 不定态传播的问题：

```
assign out = sel1 ? in1[3:0] :
             sel2 ? in2[3:0] :
             sel3 ? in3[3:0] :
                    4'b0;
```

而如果此处本来是希望生成一种并行选择的逻辑，则推荐使用 assign 语法明确地使用“与”-“或”逻辑，编写如下：

```
assign out =     ({4{sel1}} & in1[3:0])
               | ({4{sel2}} & in2[3:0])
               | ({4{sel3}} & in3[3:0]);
//使用明确的assign 语法编写的“与”-“或”逻辑一定能够保证综合生成并行选择的电路
```

问题四：

与问题三同理，Verilog 的 case 语法也会被综合成为优先级选择的电路，面积和时序均不够优化。有的 EDA 综合工具可以提供指引注释（例如 synopsys parallel_case 和 full_case）来使得综合工具能够综合出并行选择逻辑，但是这样可能会造成前后仿真不一致的严重问题，从而产生重大的 bug。因此，在实际的工程开发中：

- 应该明令禁止使用 EDA 综合工具提供的指引注释（例如 synopsys parallel_case 和 full_case）。
- 应该使用问题三推荐的等效 assign 语法编写电路。

5.3.3 其他若干注意事项

其他编码风格中的若干要点如下。

- 由于带有 reset 的寄存器面积和时序会稍微差一点，因此在数据通路上可以使用不带 reset 的寄存器，而只在控制通路上使用带 reset 的寄存器。请参见第 15.1.5 节了解更

多此低功耗设计的要诀。

- 信号名定义应该避免使用拼音，注重使用英语缩写，信号名不可定义得过长，但是也不可以定义得过短。所谓代码即注释，应该尽量从信号名中能够看出此信号的功能作用。
- Clock 和 Reset 信号应禁止被用于任何其他的逻辑功能，Clock 和 Reset 信号只能接入 DFF 作为其时钟和复位信号之用。

5.3.4 小结

上述若干点是蜂鸟 E200 代码风格最大的特点，其所推荐使用的 assign 语法和标准 DFF 例化方法能够使得任何不定态在前仿真阶段无处遁形，综合工具能够综合出很高质量的电路，综合出的电路门控时钟率也很高。

以上只是简述了蜂鸟 E200 中核心的代码风格，其他还有很多的代码风格要点在此不做赘述，感兴趣的读者可以在阅读源代码时自行体会。

5.4 蜂鸟 E200 模块层次划分

以开源的蜂鸟 E203 处理器核为例，其模块层次的划分如图 5-1 所示，要点如下。

（1）顶层的 e203_cpu_top 中仅例化两个模块，分别为 e203_cpu 和 e203_srams。

- e203_cpu 为处理器核的所有逻辑部分。
- e203_srams 为处理器核的所有 SRAM 部分（譬如 ITCM 和 DTCM 的 SRAM）。将 SRAM 和逻辑部分在层次上分开是为了方便 ASIC 实现。

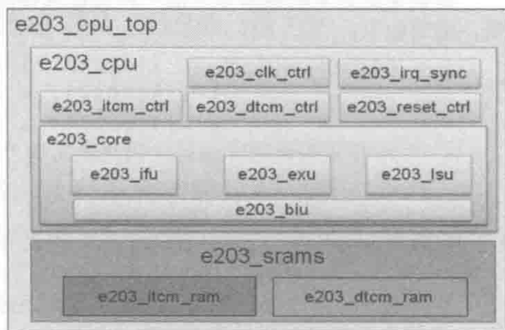


图 5-1 蜂鸟 E203 处理器核的模块层次划分

（2）逻辑顶层 e203_cpu 模块中例化的 e203_clk_ctrl 用于控制处理器各个主要组件的自动时钟门控，参见第 15.1.4 节了解更多此低功耗设计的要诀。

（3）逻辑顶层 e203_cpu 模块中例化的 e203_irq_sync 用于将外界的异步中断信号进行同步。

（4）逻辑顶层 e203_cpu 模块中例化的 e203_reset_ctrl 用于将外界的异步 reset 信号进行同步使之变成“异步置位同步释放”的复位信号。

（5）逻辑顶层 e203_cpu 模块中例化的 e203_itcm_ctrl 和 e203_dtcm_ctrl 用于控制 ITCM 和 DTCM 的访问。请参见第 11.4.4 节了解更多 ITCM 和 DTCM 的设计细节。

（6）逻辑顶层 e203_cpu 模块中例化的 e203_core 则是处理器核的主体部分，其中实现了

处理器核的主要功能。

- 取指令单元 e203_ifu，参见第 7 章了解更多细节。
- 执行单元 e203_exu，参见第 8、9、10、13 章了解更多细节。
- 存储器访问单元 e203_lsu，参见第 11 章了解更多细节。
- 总线接口单元 e203_biu，参见第 12 章了解更多细节。

5.5 蜂鸟 E200 处理器核源代码

以开源的蜂鸟 E203 处理器核为例，其处理器核的源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
      |----rtl                      // 存放 RTL 的目录
           |----e203                // E203 核和 SoC 的 RTL 目录
                |----general        // 存放一些公用的通用 RTL 代码
                |----core           // 存放 e203 Core 的 RTL 代码,
                                    // 列举主要文件如下，全部的文件列表请参见 GitHub
                     |----config.v     // 参数配置文件，参见第 4.6 节了解具体配置信息
                     |----e203_biu.v          // BIU 模块
                     |----e203_reset_ctrl.v// Core 的复位控制（Reset Control）模块
                     |----e203_clk_ctrl.v     // Core 的时钟控制（Clock Control）模块
                     |----e203_cpu_top.v      // Core 的顶层模块
                     |----e203_cpu.v          // Core 去除了 SRAM 之后的逻辑顶层模块
                     |----e203_core.v         // Core 的主体逻辑模块
                     |----e203_dtcm_ctrl.v    // DTCM 的控制模块
                     |----e203_itcm_ctrl.v    // ITCM 的控制模块
                     |----e203_exu.v          // Core 内部执行单元顶层模块
                     |----e203_ifu.v          // Core 内部取指令单元顶层模块
                     |----e203_lsu.v          // Core 内部存储器访问单元顶层模块
                     |----e203_srams.v        // Core 的所有 SRAM 的顶层模块
                     |----e203_itcm_ram.v     // ITCM 的 SRAM 模块
                     |----e203_dtcm_ram.v     // DTCM 的 SRAM 模块
```

5.6 蜂鸟 E200 处理器核配置选项

蜂鸟 E200 系列的每款处理器均有一定的可配置性。以开源的蜂鸟 E203 核为例，通过修改其目录下的 config.v 文件中的宏定义便可以实现不同的配置。请参见第 4.6 节了解具体配置信息。

5.7 蜂鸟 E200 处理器核支持的 RISC-V 指令子集

请参见附录 A 了解蜂鸟 E200 处理器核支持的 RISC-V 指令列表。

5.8 蜂鸟 E200 处理器流水线结构

蜂鸟 E200 系列处理器核的流水线结构及其各主要部分简介，请参见第 6 章。

5.9 蜂鸟 E200 处理器核顶层接口介绍

本节以开源的蜂鸟 E203 处理器核为例，介绍顶层接口信号。蜂鸟 E203 处理器的顶层模块 e203_cpu_top 的接口信号如表 5-1 所示。

表 5-1 E203 处理器核顶层接口信号列表

信号名	方向	位宽	描述
test_mode	Input	1	测试模式信号
clk	Input	1	时钟信号
rst_n	Input	1	异步复位信号，低电平有效
core_mhartid	Input	'E203_HART_ID_W	该处理器核的 HART ID 指示信号，在 SoC 集成时，可以将此信号赋予某个常数值或者信号值。请参见附录 B2.5 节了解 HART ID 的信息
pc_rtvec	Input	'E203_PC_SIZE	该输入信号用于指定处理器被 reset 后的 PC 初始值。在 SoC 层面可以通过控制此信号达到控制处理器核上电 PC 初始值的效果
ext_irq_a	Input	1	外部中断，来自 PLIC 模块。请参见第 13 章了解 RISC-V 中断和 PLIC 的相关信息
sft_irq_a	Input	1	软件中断，来自 CLINT 模块。请参见第 13 章了解 RISC-V 中断和 CLINT 的相关信息
tmr_irq_a	Input	1	计时器中，断来自 CLINT 模块。请参见第 13 章了解 RISC-V 中断和 CLINT 的相关信息
core_wfi	Output	1	该输出信号如果为高电平，则指示此处理器核处于执行 WFI 指令之后的休眠状态。请参见第 15.3.2 节了解 WFI 指令进入低功耗休眠状态的信息

续表

信号名	方向	位宽	描述
tm_stop	Output	1	此信号用于固定与 SoC 中的 CLINT 相连接。请参见第 13.5.5 节了解更多 CLINT 模块的细节 该输出信号的值来自于蜂鸟 E200 自定义的 CSR 寄存器 mcounterstop 中的 TIMER 域。请参见附录 B3.1 节了解 mcounterstop 寄存器更多信息
ext2itcm_icb_cmd_valid ext2itcm_icb_cmd_ready ext2itcm_icb_cmd_addr ext2itcm_icb_cmd_read ext2itcm_icb_cmd_wdata ext2itcm_icb_cmd_wmask ext2itcm_icb_rsp_valid ext2itcm_icb_rsp_ready ext2itcm_icb_rsp_err ext2itcm_icb_rsp_rdata	此组信号为 ITCM 外部接口的 ICB 总线信号，参见第 11.4.4 节了解有关 ITCM 的信息和其外部接口信息		
ext2dtcm_icb_cmd_valid ext2dtcm_icb_cmd_ready ext2dtcm_icb_cmd_addr ext2dtcm_icb_cmd_read ext2dtcm_icb_cmd_wdata ext2dtcm_icb_cmd_wmask ext2dtcm_icb_rsp_valid ext2dtcm_icb_rsp_ready ext2dtcm_icb_rsp_err ext2dtcm_icb_rsp_rdata	此组信号为 DTCM 外部接口的 ICB 总线信号，参见第 11.4.4 节了解有关 DTCM 的信息和其外部接口信息		
ppi_icb_cmd_valid ppi_icb_cmd_ready ppi_icb_cmd_addr ppi_icb_cmd_read ppi_icb_cmd_wdata ppi_icb_cmd_wmask ppi_icb_rsp_valid ppi_icb_rsp_ready ppi_icb_rsp_err ppi_icb_rsp_rdata	此组信号为私有外设接口的 ICB 总线信号，参见第 12 章了解有关 ICB 的信息和私有外设接口信息		

续表

信号名	方向	位宽	描述
fio_icb_cmd_valid fio_icb_cmd_ready fio_icb_cmd_addr fio_icb_cmd_read fio_icb_cmd_wdata fio_icb_cmd_wmask fio_icb_rsp_valid fio_icb_rsp_ready fio_icb_rsp_err fio_icb_rsp_rdata	此组信号为快速 IO 接口的 ICB 总线信号，参见第 12 章了解有关 ICB 的信息和快速 IO 接口信息		
mem_icb_cmd_valid mem_icb_cmd_ready mem_icb_cmd_addr mem_icb_cmd_read mem_icb_cmd_wdata mem_icb_cmd_wmask mem_icb_rsp_valid mem_icb_rsp_ready mem_icb_rsp_err mem_icb_rsp_rdata	此组信号为系统存储接口的 ICB 总线信号，参见第 12 章了解有关 ICB 的信息和系统存储接口信息		
clint_icb_cmd_valid clint_icb_cmd_ready clint_icb_cmd_addr clint_icb_cmd_read clint_icb_cmd_wdata clint_icb_cmd_wmask clint_icb_rsp_valid clint_icb_rsp_ready clint_icb_rsp_err clint_icb_rsp_rdata	此组信号为 CLINT 接口的 ICB 总线信号，参见第 12 章了解有关 ICB 的信息和 CLINT 接口信息		
plic_icb_cmd_valid plic_icb_cmd_ready plic_icb_cmd_addr plic_icb_cmd_read plic_icb_cmd_wdata plic_icb_cmd_wmask plic_icb_rsp_valid plic_icb_rsp_ready plic_icb_rsp_err plic_icb_rsp_rdata	此组信号为 PLIC 接口的 ICB 总线信号，参见第 12 章了解有关 ICB 的信息和 PLIC 接口信息		

续表

信号名	方向	位宽	描述
dbg_irq_r cmt_dpc cmt_dpc_ena cmt_dcause cmt_dcause_ena wr_dcsr_ena wr_dpc_ena wr_dscratch_ena wr_csr_nxt dcsr_r dpc_r dscratch_r	此组信号用于固定与 SoC 中的调试模块（Debug Module）连接，普通用户无须关注此部分接口。请参见第 14 章了解更多调试模块的信息		

5.10 总结

本章仅对蜂鸟 E200 处理器核的设计进行宏观的介绍，帮助读者从整体认识蜂鸟 E200 处理器的设计要诀。请读者继续阅读后续章节，有针对性地学习处理器不同部分的细节，以透彻地了解蜂鸟 E200 处理器核的设计。

第6章 流水线不是流水账——蜂鸟E200流水线介绍



本章将讨论处理器的一个重要的基础知识——“流水线”，和本章标题流水账无任何关联，仅取一个谐音而已。

熟悉计算机体系结构的读者一定知道，言及处理器微架构，几乎必谈其流水线。处理器的流水线结构是处理器微架构最基本的一个要素，犹如汽车底盘对于汽车一般具有基石性的作用，它承载并决定了处理器其他微架构的细节。本章将简要介绍处理器的一些常见流水线结构，并介绍蜂鸟 E200 处理器核的流水线微架构。

6.1 处理器流水线概述

6.1.1 从经典的五级流水线说起

流水线的概念来源于工业制造领域，以汽车装配为例来解释流水线的工作方式，假设装配一辆汽车需要 4 个步骤。

- 第 1 步：冲压，制作车身外壳和底盘等部件。
- 第 2 步：焊接，将冲压成形后的各部件焊接成车身。
- 第 3 步：涂装，将车身等主要部件清洗、化学处理、打磨、喷漆和烘干。
- 第 4 步：总装，将各部件（包括发动机和向外采购的零部件）组装成车。

同时需要对应冲压、焊接、涂装和总装 4 项工作的工人。最简单的方法是一辆汽车依次经过上述 4 个步骤装配完成之后，下一辆汽车才开始进行装配，最早期的工业制造就是采用的这种原始的方式，即同一时刻只有一辆汽车在装配。不久之后人们发现，一辆汽车在某个时段中进行装配时，其他 3 个工人都处于闲置状态，显然这是对资源的极大浪费，于是思考出能有效利用资源的新方法。即在第一辆汽车经过冲压进入焊接工序时，立刻开始进行第二辆汽车的冲压，而不是等到第一辆汽车经过全部 4 个工序后才开始。这样在后续生产中就能够保证 4 个工人一直处于运行状态，不会造成人员的闲置。这样的生产方式就好似流水川流不息，因此被称为流水线。

计算机体系结构教材中被提及最多的经典 MIPS 五级流水线如图 6-1 所示。在此流水线中一条指令的生命周期分为如下步骤。

（1）取指

- 指令取指（Instruction Fetch）是指将指令从存储器中读取出来的过程。

（2）译码

- 指令译码（Instruction Decode）是指将从存储器中取出的指令进行翻译的过程。经过译码之后得到指令需要的操作数寄存器索引，可以使用此索引从通用寄存器组（Register File，Regfile）中将操作数读出。

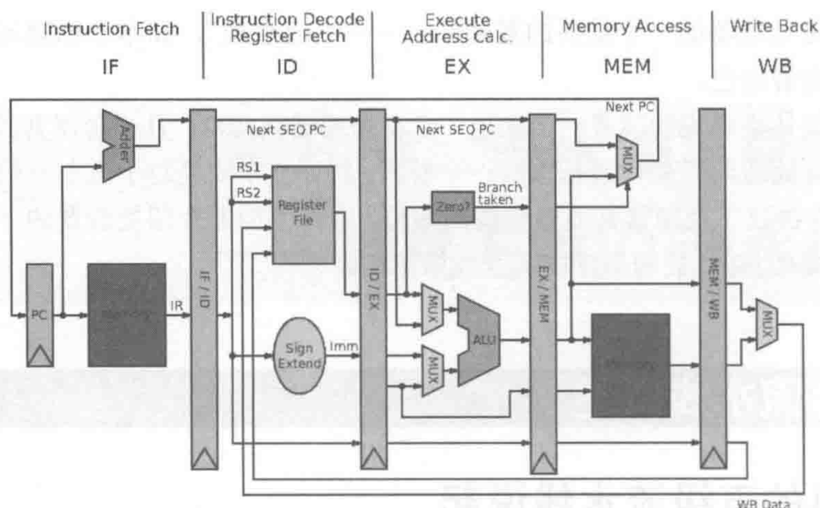


图 6-1 MIPS 五级流水线结构图

（3）执行

- 指令译码之后所需要进行的计算类型都已得知，并且已经从通用寄存器组中读取出了所需的操作数，那么接下来便进行指令执行（Instruction Execute）。指令执行是指对指令进行真正运算的过程。譬如，如果指令是一条加法运算指令，则对操作数进行加法操作；如果是减法运算指令，则进行减法操作。
- 在“执行”阶段的最常见部件为算术逻辑部件运算器（Arithmetic Logical Unit，ALU），作为实施具体运算的硬件功能单元。

（4）访存

- 存储器访问指令往往是指令集中最重要的指令类型之一，访存（Memory Access）是指存储器访问指令将数据从存储器中读出，或者写入存储器的过程。

（5）写回

- 写回（Write-Back）是指将指令执行的结果写回通用寄存器组的过程。如果是普通运算指令，该结果值来自于“执行”阶段计算的结果；如果是存储器读指令，该结果来自于“访存”阶段从存储器中读取出来的数据。

在工业制造中采用流水线可以提高单位时间的生产量，同样在处理器中采用流水线设计也有助于提高处理器的性能。以上述的五级流水线为例，由于前一条指令在完成了“取指”进入“译码”阶段后，下一条指令马上就可以进入“取指”阶段，依次类推。如图 6-2 所示，如果流水线没有停顿，理论上可以取得每个时钟周期都完成一条指令的性能。

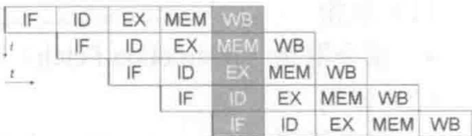


图 6-2 MIPS 五级流水线运行图

6.1.2 可否不要流水线——流水线和状态机的关系

如上一节所述，在绝大多数的情况下，言及处理器微架构，几乎必谈流水线。那么，我们能否挑战一下权威提出一个有意思的问题：处理器难道就一定需要流水线吗？可否不要流水线呢？

在回答这个问题之前，我们先探讨下流水线的本质。

- 流水线并不限于处理器设计，在所有的 ASIC 电路实现中都广泛采用流水线的思想。流水线本质上可以理解为一种以面积换性能（Trade Area for Performance）、以空间换时间（Trade Space for Timing）的手段。以五级流水线为例，其增加了 5 组寄存器，每一个流水线级数内部都有各自的组合逻辑数据通路，彼此之间没有复用资源，因此其面积开销是比较大的。但是由于可以让不同的流水线级数同时做不同的事情，而达到流水的效果，提高了性能，优化了时序，增加了吞吐率。
- 与流水线相对应的另外一种策略是状态机，状态机是流水线的“取反”，同样在所有的 ASIC 电路实现中都广泛采用。状态机本质上可以理解为是一种以性能换面积（Trade Performance for Area）、以时间换空间（Trade Timing for Space）的手段。
- “流水线”和“状态机”的关系，还有一种说法称之为“展开”和“折叠”的关系。本质上都是一种电路设计时，选择侧重时间（性能）还是空间（面积）的一种取舍。

通过上述分析，假设处理器不采用流水线，而是使用一个状态机来完成，则需要多个时钟周期才能完成一条指令的所有操作，每个时钟周期完成状态机的一个状态（分别为取指、译码、执行、访存和写回）。通过使用状态机，可以省掉上述流水线中的寄存器开销，还可以复用组合逻辑数据通路，因此面积开销比较小。但是每条指令都需要 5 个周期才能完成，吞吐率和性能很差。

谈及此处，就不得不提及 8 位单片机时代的“传奇老炮儿”8051 内核，早期原始的 8051 内核微架构就是采用了类似状态机的实现方式而不是流水线。回到最开始我们提出的问题，处理器可否不要流水线？答案是：当然可以， 8051 内核就没有流水线。

所以说从功能能上来讲，处理器完全可以不使用流水线，而使用状态机的方式来实现，只不过由于这种方式性能比较差，在现代处理器设计中比较罕见而已。

6.1.3 深处种菱浅种稻，不深不浅种荷花——流水线的深度

上一节讨论了流水线能够提高处理器的性能，基本上是现代处理器的必备要素。那么流水线的级数（又称深度）多少最好呢？要回答这个问题，就需要了解流水线的深浅各自的优劣。此处有一个常见面试题：处理器的流水线是否越深越好？在此我们给出答案。

- 早期的经典流水线是五级流水线，分别为取指、译码、执行、访存和写回。现代的处理器往往具有极深的流水线级数，譬如高达十几级，或者二十几级的深度。流水线就像一根黄瓜，切 2 刀下去得到的每一截长度和切 20 刀下去得到的每一截长度肯定是不一样的。流水线的级数越多，意味着流水线被切得越细，每一级流水线内容纳的硬件逻辑便越少。熟悉数字同步电路设计的读者应该比较熟悉，在两级寄存器（每一级流水线由寄存器组成）之间的硬件逻辑越少，则意味能够运行到更高的主频。因此，现代的处理器流水线极深主要是由于处理器追求高频的指标所驱使。高端的 ARM Cortex-A 系列由于有十几级的流水线，所以能够运行到高达 2GHz 的主频，而 Intel 的 x86 处理器甚至采用几十级的流水线深度将主频推到 3～4GHz 的高度。主频越高也意味着流水线的吞吐率越高，从而性能越高，这是流水线加深的正面意义。
- 由于每一级流水线都由寄存器组成，更多的流水线级数要消耗更多的寄存器，以及更多的面积开销。这是流水线加深的负面意义。
- 由于每一级流水线需要进行握手，流水线最后一级的反压信号可能会一直串扰到最前一级造成严重的时序问题，需要使用一些比较高级的技巧来解决此类反压时序问题（第 6.3 节将进一步论述）。这是流水线加深的负面意义。
- 较深的处理器流水线还有一个问题，那就是由于在流水线的取指令阶段无法得知条件跳转的结果是到底跳还是不跳，因此只能进行预测，而到了流水线的末端才能够通过实际的运算得知该分支是真的该跳还是不该跳。如果发现真实的结果（譬如该跳）与之前预测的结果（譬如预测为不跳）不相符，则意味着预测失败，需要将所有预取的错误指令流全部丢弃掉。重新取正确的指令流，这个过程叫作“流水线冲刷（Pipeline Flush）”。虽然可以使用分支预测器来保证前期的分支预测尽可能准确，但是也无法做到万无一失。那么，流水线的深度越深，意味着已经预取了更多的错误指令流，需要将其全部抛弃然后重启，不仅白白浪费了功耗，还造成了性能的损失。流水线越深，则意味着浪费和损失越严重；流水线越浅，则浪费和损失越少。这是流水线加深的另一个主要的负面意义。

综上，所谓“深处种菱浅种稻，不深不浅种荷花”，流水线的不同深度皆有其优缺点，需要根据不同的应用背景进行合理的选择。

由于处理器流水线深浅的不同优劣，根据不同的应用场景，当今处理器的流水线深度在向着两个不同的极端发展，一方面级数越来越深，另一方面又越来越浅，下面结合不同的商用处理器例子予以探讨。

6.1.4 向上生长——越来越深的流水线

现代的高性能处理器相比最早期的处理器明显存在着流水线越来越深的现象，其驱动因

素很简单，那就是追求更高的主频以获取更高的吞吐率和性能。

以最知名的 ARM Cortex-A 系列处理器 IP 为例，Cortex-A7 主打的低功耗前提下的能效比，其流水线级数为 8 级；而 Cortex-A15 主打高性能，其流水线深度为 15 级。

当然流水线越来越深也需有其限度，曾有某些商业处理器产品一味地追求极端流水线深度（达到几十级）反而遭遇失败的例子，目前最新的 Intel 处理器和 ARM 高性能 Cortex-A 系列处理器的流水线深度都在十几级的范围。

6.1.5 向下生长——越来越浅的流水线

现代低功耗处理器的另外一个趋势也存在着流水线越来越浅的现象，其驱动因素同样很简单，那就是在性能够用的前提下追求极低的功耗。

以最知名的 ARM Cortex-M 系列处理器 IP 为例，2004 年发布的 Cortex-M3 处理器核的流水线级数只有三级，2009 年发布的 Cortex-M0 处理器核的流水线级数也只有三级。而 2012 年发布的 Cortex-M0+处理器核的流水线级数反而只有二级，流水线级数变得越来越少，因此 ARM 也宣传 Cortex-M0+处理器核为世界上能效比最高的处理器核。

二级的流水线深度似乎已经浅到底了，那是不是接下来要发布只有一级深度的流水线了？当深度变为 1 之后也就谈不上流水线了，其整体也就变成一个单周期的组合逻辑。在众多的计算机体系结构教学案例中我们确实见到过很多流水线深度为 1 的处理器核，从功能上来说其仍然可以完成处理器的所有功能，只不过主频相当之低。有没有商业的处理器核真的只有一级的流水线深度，作者在此无法确定，但是别忘了第 6.1.2 节中提及的 8051 内核，别说一级的流水线深度，连状态机都用上了。

6.1.6 总结

至此，本节简单重温了处理器流水线的相关概念。这些概念是计算机体系结构中很基础的知识，本书限于篇幅在此不做过多赘述。若完全无处理器知识背景的读者仍无法理解，可以参见维基百科上的词条网页（请在维基百科中搜索“Classic_RISC_pipeline”）了解更多信息。

6.2 处理器流水线中的乱序

处理器中发射、派遣、执行、写回的顺序也是处理器微架构设计中非常重要的一环，可以衍生出“顺序”和“乱序”的概念，本书将在第 8 章对此进行专门论述。

6.3 处理器流水线中的反压

注意：此节关于解决反压的技术方法描述可能过于晦涩，需要对于 ASIC 电路设计有较深的经验方能理解，初学者可以忽略此节无须深究。

第 6.1.4 节中提到，流水线越深，由于每一级流水线需要进行握手，流水线最后一级的反压信号可能会一直串扰到最前一级造成严重的反压（Back-pressure）时序问题，需要使用一些比较高级的技巧来解决这些时序问题。在现代处理器设计中，通常有如下 3 种方法。

- 取消握手：此方法能够杜绝反压的发生，时序表现非常好。但是取消握手，即意味着流水线中的每一级并不会与其下一级进行握手，可能会造成功能错误或者指令丢失。因此这种方法往往需要配合其他的机制，譬如重执行（Replay）、预留大缓存等。简而言之，此方法比较激进，辅以一系列其他的配置机制，硬件总体的复杂度会比较大，只有在一些非常高级的处理器设计中才会用到。
- 加入乒乓缓存：加入乒乓缓存（Ping-pong Buffer）是一种用面积换时序的方法，也是在解决反压的最简单方法。通过使用乒乓缓存（有两个表项）替换掉普通的一级流水线（只有一个表项），可以使得此级流水线向上一级流水线的握手接收信号仅关注乒乓缓存中是否有一个以上有空的表项即可，而无需将下一级的握手接收信号串扰至上一级。
- 加入前向旁路缓存：加入前向旁路缓存（Forward Bypass Buffer）也是一种用面积换时序的方法，是在解决反压时的一种非常巧妙的方法。旁路缓存仅只有一个表项，由于增加了这一个额外的缓存表项，可以将后向的握手信号时序路径砍断，但是对前向路径不受影响，因此可以广泛使用于握手接口。蜂鸟 E200 即于设计中采用此方法，有效地解决了多处反压造成的时序瓶颈。

以上解决反压的技术方法，不仅在处理器设计中能够用到，而且在普通的 ASIC 电路设计中也会经常用到。

6.4 处理器流水线中的冲突

处理器的流水线设计中另外一个问题便是流水线中的冲突（Hazards），主要分为资源冲突和数据冲突。

6.4.1 流水线中的资源冲突

资源冲突是指流水线中硬件资源的冲突，最常见的是运算单元的冲突，譬如除法器需要

多个时钟周期才能完成运算。因此在前一个除法指令完成运算之前，新的除法指令如果也需要除法器，则会存在着资源冲突。在处理器的流水线中硬件资源冲突种类还有较多，在此不一一赘述。解决资源冲突可以通过复制硬件资源或者流水线停顿等待硬件资源的方法解决。

6.4.2 流水线中的数据冲突

数据冲突是指不同的指令之间的操作数存在着数据相关性造成的冲突，常见的数据相关性如下。

- WAR（Write-After-Read）相关性，又称先读后写相关性：表示"后序执行的指令需要写回的结果寄存器索引"与"前序执行的指令需要读取的源操作数寄存器索引"相同造成的数据相关性。因此从理论上来讲，在流水线中"后序指令"一定不能比和它有 WAR 相关性的"前序指令"先执行，否则"后序指令"先写回了结果至通用寄存器组中，"前序指令"再读取操作数时，就会读到错误的数值。
- WAW（Write-After-Write）相关性，又称先写后写相关性：表示"后序执行的指令需要写回的结果寄存器索引"与"前序执行的指令需要写回的结果寄存器索引"相同造成的数据相关性。因此从理论上来讲，在流水线中"后序指令"一定不能比和它有 WAW 相关性的"前序指令"先执行，否则"后序指令"先写回了结果至通用寄存器组中，"前序指令"再写回结果至通用寄存器组中就会将其覆盖。
- RAW（Read-After-Write）相关性，又称先写后读相关性：表示"后序执行的指令需要读取的源操作数寄存器索引"与"前序执行的指令需要写回的结果寄存器索引"相同造成的数据相关性。因此从理论上来讲，在流水线中"后序指令"一定不能比和它有 RAW 相关性的"前序指令"先执行，否则"后序指令"便会从通用寄存器组中读回错误的源操作数。

以上的 3 种相关性中，RAW 属于真数据相关。

解决数据冲突的常见方法如下。

（1）WAW 和 WAR 可以通过寄存器重命名的方法将相关性去除，从而无须担心其执行顺序。

- 寄存器重命名技术在 Tomasulo 算法中通过保留站和 ROB（Re-Order Buffer）完成，或者采用纯物理寄存器（而不用 ROB）的方式完成。有关 ROB 和纯物理寄存器的作用和信息在第 8 章中对"指令发射、派遣、执行、写回的顺序和常见策略"的介绍中将进一步论述。

（2）之所以称 RAW 为真数据相关，是因为其没有办法通过寄存器重命名的方法将相关性去除。一旦产生 RAW 相关性，后序的指令一定要使用和它有 RAW 数据相关性的前序指令执行完成的结果，从而造成流水线的等待停顿。为了能够尽可能减少流水线停顿带来的性

能损失，可以使用“动态调度”的方法。动态调度的思想本质上可以归结于以下方面。

- 一方面采用数据旁路传播（Data Bypass and Forward）技术，尽可能让前序指令的计算结果更快地旁路传播给后序相关指令的操作数。
- 另一方面尽可能地让后序相关指令在等待的过程中不阻塞流水线，而让其他无关的指令继续顺利执行。
- 早期的 Tomasulo 算法中通过保留站可以达到这两方面的功效，但是保留站由于保存了操作数，无法做到很大的深度（否则面积和时序的开销巨大）。
- 最新的高性能处理器普遍采用在每个运算单元前配置乱序发射队列（Issue Queue）的方式，发射队列仅追踪 RAW 相关性，而并不存放操作数，因此可以做到很深（譬如 16 个表项）。在发射队列中的指令一旦相关性解除之后，再从发射队列中发射出来读取物理寄存器组（Physical Register File），然后发送给运算单元开始计算。

有关处理器的数据相关性问题和包括动态调度技术在内的解决方法，如果阐述清楚几乎可以单独成书，本书限于篇幅只能予以简述。有关 Tomasulo 算法的细节请参见维基百科词条网页（请在维基百科中搜索“Tomasulo_algorithm”），和有关物理寄存器重命名的细节请参见维基百科词条网页（请在维基百科中搜索“Register_renaming”），感兴趣的读者可以自行查阅。

6.5 蜂鸟 E200 处理器的流水线

6.5.1 流水线总体结构

蜂鸟 E200 处理器的总体结构如图 6-3 所示，要点如下。

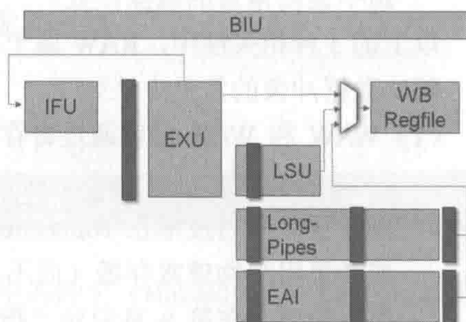


图 6-3 蜂鸟 E200 处理器核的流水线结构

（1）流水线的第一级为“取指（由 IFU 完成）”。

（2）蜂鸟 E200 处理器核很难严谨界定它的完整流水线级数为几级，原因如下。

- “译码（由 EXU 中完成）”“执行（由 EXU 中完成）”和“写回（由 WB 完成）”均处于同一个时钟周期，位于流水线的第二级。
- 而“访存（由 LSU 完成）”阶段处于 EXU 之后的第三级流水线，但是 LSU 写回的结果仍然需要通过 WB 模块写回通用寄存器组（Register File，Regfile）。

- 因此严格来讲，蜂鸟 E200 是一个变长流水线结构。

由于蜂鸟 E200 处理器核的流水线的按序主体是位于第一级的“取指”和位于第二级的“执行”和“写回”，因此我们非严谨地定义蜂鸟 E200 处理器核的流水线深度为二级。

本书后续章节将会具体介绍流水线中的各个主要部分和单元。

- 有关取指（IFU 单元）的实现细节，请参见第 7 章。
- 有关执行（EXU 单元）和长指令（Long Pipes）的实现细节，请参见第 8、9 章。
- 有关写回（WB 单元）的实现细节，请参见第 10 章。
- 有关访存（LSU 单元）的实现细节，请参见第 11 章。
- 取指和访存若需要访问外部存储器，均需通过 BIU 单元完成，有关 BIU 单元的实现细节，请参见第 12 章。
- 有关 EAI 协处理器的更多信息，请参见第 16 章。

6.5.2 流水线中的冲突

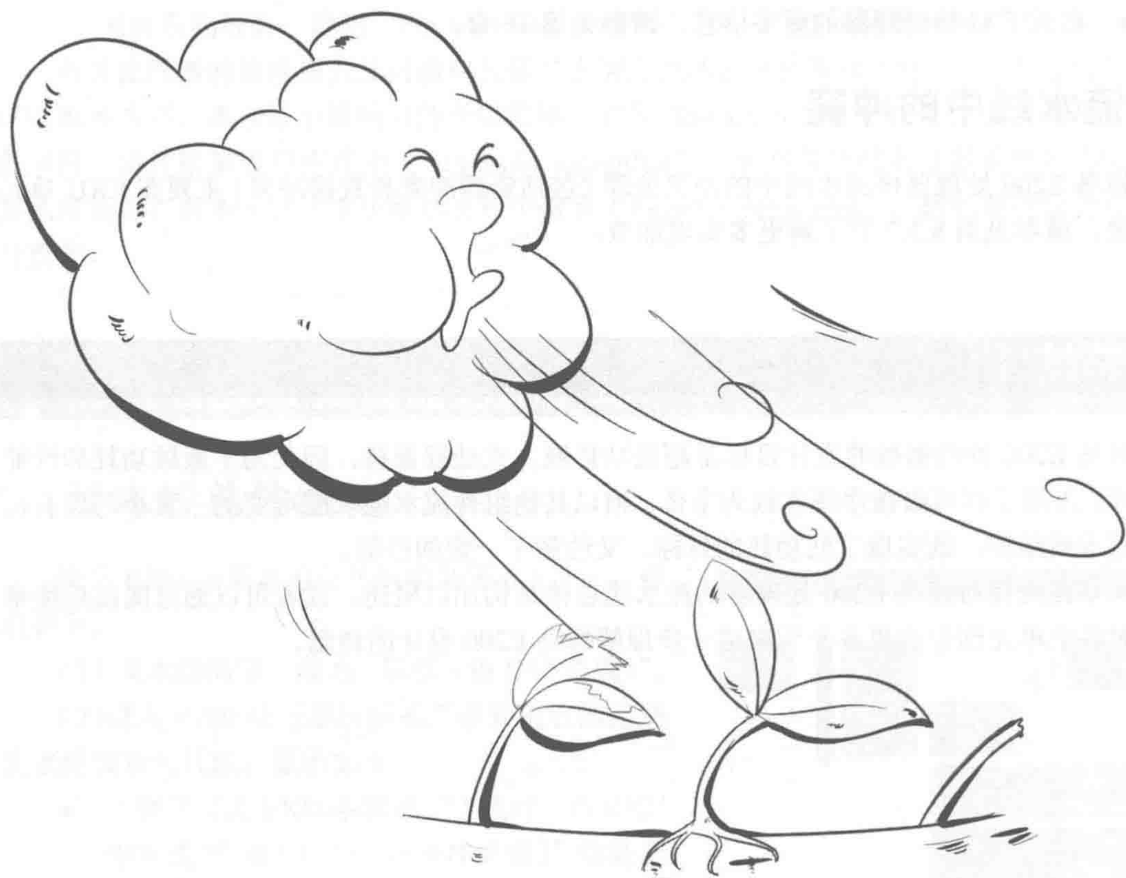
蜂鸟 E200 处理器核流水线中的冲突处理（包括资源冲突和数据冲突）主要在 EXU 单元中解决，请参见第 8.3.7 节了解更多实现细节。

6.6 总结

蜂鸟 E200 处理器核的设计目标是超低功耗嵌入式处理器核，因此为了兼顾功耗和性能的目标，采用了以两级按序流水线为主体，辅以其他组件流水线长度可变的一套小巧而有特点的流水线结构，既实现了低功耗的目标，又达到了一定的性能。

本章在此仅对蜂鸟 E200 处理器的流水线总体结构加以概述，读者可以通过阅读后续章节中对各个单元部分的更多介绍来进一步理解蜂鸟 E200 设计的精髓。

第 7 章　万事开头难吗
——一切从取指令开始



万事开头难

在上一章我们介绍了处理器流水线的总体结构，处理器流水线中第一步是“取指”。所谓万事开头难，本章将简要介绍处理器的“取指”功能，并介绍蜂鸟 E200 处理器核取指单元（Instruction Fetch Unit，IFU）的微架构和源码分析。

7.1 取指概述

7.1.1 取指特点

处理器执行的汇编指令流示例如图 7-1 所示。每条指令在存储器空间中所处的地址称为它的指令 PC（Program Counter）。取指（Instruction Fetch）是指处理器核将指令从存储器中读取出来的过程（按照其指令 PC 值对应的存储器地址）。

取指的终极目标是以最快的速度且连续不断地从存储器中取出指令供处理器核执行，核心要点是“快”和“连续不断”。为了能够达到这两个目标，我们以图 7-1 中的示例先分析常规 RISC 架构汇编指令流的特点。

- 对于非分支跳转指令，如图 7-1 所示，从 PC 值为 0x80002150 处至 PC 值为 0x8000215e 处之间的指令都是非分支跳转指令，处理器需要按顺序执行这些指令，指令 PC 值逐条指令连续增加。因此处理器在取指的过程中可以按顺序从存储器中读取出指令。
- 对于分支跳转指令，处理器执行了这条指令后，如果该跳转指令的条件成立需要发生跳转，则会跳转至另外一个不连续的PC值处。如图7-1 所示，从PC值为0x80002160 处的 bne 指令，如果操作数 a6 和 a3 寄存器中的值不相等，则需要发生跳转去执行 PC 为 0x80002150 处的指令。因此处理器在取指的过程中，理论上也需要从新的 PC 值对应的存储器地址读取出指令。
- 指令的编码长度可以不相等，如图 7-1 所示，有的指令编码宽度是 16 位的，而有的指令编码宽度是 32 位的。对于宽度为 32 位的指令，其对应的 PC 地址可能与 32 位地址边界不对齐，譬如图 7-1 中 PC 值为 0x8000217a 处的 32 位指令所处的存储器地址（0x8000217a）便与 32 位地址边界不对齐（无法被 4 整除）。

综上，结合 RISC 架构汇编指令流的特点，处理器需要以“快”和“连续不断”的标准从存储器中取出指令，就需要能够做到如下性能。

- 对于非分支跳转指令，能够连续不断地按顺序将其从存储器中快速读取出来，即便是地址不对齐的 32 位指令，也最好能够连续不断地每个周期读出一条完整指令。
- 对于分支跳转指令，能够快速地判定其是否需要跳转。如果需要跳转，则从新的 PC 地址处快速取出指令，即便是地址不对齐的 32 位指令，也最好能够一个周期读出一条完整指令。

指令PC　指令编码　汇编指令

```
80002150:   429c          lw   a5,0(a3)
80002152:   42c8          lw   a0,4(a3)
80002154:   468c          lw   a1,8(a3)
80002156:   06b1          addi     a3,a3,12
80002158:   97aa          add a5,a5,a0
8000215a:   97ae          add a5,a5,a1
8000215c:   c21c          sw   a5,0(a2)
8000215e:   0611          addi     a2,a2,4
80002160:   fed818e3   bne a6,a3,80002150 <main+0x68>
80002164:   474c          lw   a1,12(a4)
80002166:   4f10          lw   a2,24(a4)
80002168:   4785          li   a5,1
8000216a:   0040          addi     s0,sp,4
8000216c:   97ae          add a5,a5,a1
8000216e:   86a2          mv   a3,s0
80002170:   97b2          add a5,a5,a2
80002172:   c29c          sw   a5,0(a3)
80002174:   1028          addi     a0,sp,40
80002176:   0711          addi     a4,a4,4
80002178:   0691          addi     a3,a3,4
8000217a:   00e50c63   beq a0,a4,80002192 <main+0xaa>
8000217e:   431c          lw   a5,0(a4)
80002180:   474c          lw   a1,12(a4)
```

图 7-1　汇编指令流示例

下文将分别予以论述。

7.1.2　如何快速取指

为了能够以更“快”的速度从存储器中取出指令，首先需要保证存储器的读延迟越小越好。不同的存储器类型有不同的延迟，片外的 DDR 存储器或者 Flash 存储器可能需要几十个周期的延迟，片上的 SRAM 也可能要几个周期的延迟。为了能够使得处理器核以最快的速度取指，通常使用 ITCM 和 I-Cache 的方法。

（1）ITCM（Instruction Tightly Coupled Memory）

- 指令紧耦合存储器，是指配置一段较小容量（一般几十 KB）的存储器（通常使用 SRAM），用于存储指令，且在物理上离处理器核很近而专属于处理器核，因此能够取得很小的访问延迟（通常一个时钟周期）。
- ITCM 的优点是实现非常简单，容易理解，且能保证实时性。
- ITCM 的缺点是由于使用地址区间寻址，因此无法像缓存（Cache）那样映射无限大的存储器空间；同时为了保证足够小的访问延迟，无法将容量做到很大（否则无法一个时钟周期访问出来 SRAM 或芯片无法容纳过大的 SRAM），因此 ITCM 只能用于存放容量大小有限的关键程序指令。

（2）I-Cache（Instruction Cache）

- 指令缓存，是指利用软件程序的时间局部性和空间局部性，将容量巨大的外部指令存储器空间动态映射到容量有限的指令缓存中，可以将访问指令存储器的平均延迟降低到最小。
- 由于缓存的容量是有限的，因此访问缓存存在着相当大的不确定性。一旦缓存不命中（Cache Miss），则需要从外部的存储器中存取数据，造成较长的延迟。在实时性

要求高的场景中，处理器的反应速度必须有最可靠的实时性。如果使用了缓存，则无法保证这一点。关于缓存实时性的论述请参见第11.1.1节。

- 大多数极低功耗处理器应用的场景都应用于实时性较高的场景，因此更加倾向于使用延迟确定的ITCM。
- 此外，缓存几乎可以认为是处理器微架构中最复杂的部分，请参见第11.1.1节了解更多信息。有关缓存的相关知识以及设计技巧的介绍几乎可以单独成书，本文限于篇幅在此不做赘述，感兴趣的读者可以参见维基百科上关于缓存的词条网页（请在维基百科中搜索“CPU_cache”）。

7.1.3 如何处理非对齐指令

如第7.1.1节中所述，“连续不断”是处理器取指的另一个目标。如果每一个时钟周期都能够取出一条指令，就可以源源不断地为处理器后续执行提供指令流，而不会出现空闲的时钟周期。

但是，不管是从I-Cache，还是从ITCM中取指令，当处理器取指遇到了一条地址非对齐的指令，则会为“连续不断”取指造成困难，因为ITCM和I-Cache的存储单元往往使用SRAM，而SRAM的读端口往往具有固定宽度。以宽度为32位的SRAM为例，其一个时钟周期只能读出一个（地址与32位对齐）32位的数据。假设一条32位长的指令处于地址不对齐的位置，则意味着需要分两个时钟周期读出两个32位的数据，然后各取其一部分进行拼接成为真正需要的32位指令，这样就需要花费至少2个时钟周期才能够取出一条指令来。

如何才能使得处理器对于非对齐的指令都能一个周期将其取出？对于普通指令和分支跳转指令需要分别论述。

1. 普通指令非对齐

对于普通指令的按顺序取指（地址连续增长）情形，可以使用剩余缓存（Leftover Buffer）保存上次取指令后没有用完的比特位，供下次使用。假设从ITCM中取出一个32位的指令字，但是只用到了它的低16位，这种情形可能是由于两种原因造成的。

- 只需要使用此次取出的32位中的低16位和上一次取出的高16位组成了一条32位指令。
- 这个指令长度本身就是16位宽，因此只需要取出的低16位。

那么对于此次没有使用到的高16位，则可以暂存于剩余缓存中，待下个周期取出下一个32位的指令字之后，就可以马上拼接出新的完整32位指令字。

2. 分支跳转指令非对齐

对于分支跳转指令而言，如果跳转的目标地址与32位地址边界不对齐，且需要取出一个32位的指令字，上述剩余缓存也无济于事了（因为剩余缓存只有在按顺序取指时，才能提前预存上次没有用完的指令字）。对此，常见的实现方式是使用多体（Bank）化的SRAM进行指令存储。以常见的奇偶交错方式为例，使用两块32位宽的SRAM交错地进行存储，

两个连续的 32 位指令字将会被分别存储在两块不同的 SRAM 中。这样对于地址不与 32 位对齐的指令，则一个周期可以同时访问两块 SRAM 取出两个连续的 32 位指令字，然后各取其一部分进行拼接成为真正需要的 32 位指令。

7.1.4 如何处理分支指令

1. 分支指令类型

在论述如何处理分支指令之前，有必要对 RISC 架构处理器的分支指令类型进行介绍，常见的分支指令类型如下。

（1）无条件跳转/分支（Unconditional Jump/Branch）指令，是指（无需判断条件）一定会发生跳转的指令，而按照跳转的目标地址计算方式，还分为以下两种情况。

- 无条件直接跳转/分支（Unconditional Direct Jump/Branch）指令，此处的“直接”是指跳转的目标地址从指令编码中的立即数可以直接计算而得。

  以 RISC-V 架构中的 jal（jump and link）指令为例，便属于无条件直接跳转指令。jal 指令的汇编示例如“jal x5, offset”，jal 使用编码在指令字中的 20 位立即数（有符号数）作为偏移量（offset）。该偏移量乘以 2，然后与当前指令所在的地址相加，生成得到最终的跳转目标地址。

- 无条件间接跳转/分支（Unconditional Indirect Jump/ Branch）指令，此处的“间接”是指跳转的目标地址需要从寄存器索引的操作数中计算出来。

  以 RISC-V 架构中的 jalr（jump and link-register）指令为例，便属于无条件间接跳转指令。jalr 指令的汇编示例如“jalr x1, x6, offset”，jalr 与 jal 的不同之处在于 jalr 使用编码在指令字中的 12 位立即数（有符号数）作为偏移量（offset），与 jalr 的另外一个寄存器索引的操作数（基地址寄存器）相加得到最终的跳转目标地址。

（2）带条件跳转/分支（Conditional Jump/ Branch），是指需要判断条件而决定是否发生跳转的指令，同样按照跳转的目标地址计算方式，还分为以下两种情况。

- 带条件直接跳转/分支（Conditional Direct Jump/Branch）指令，此处的“直接”是指跳转的目标地址从指令编码中的立即数可以直接计算而得。

  以 RISC-V 架构为例，其有 6 条带条件分支指令（Conditional Branch），这种带条件的分支指令跟普通的运算指令一样直接使用两个整数操作数，然后对其进行比较。如果比较的条件满足，则进行跳转。

- 带条件间接跳转/分支（Conditional Indirect Jump/ Branch）指令，此处的“间接”是指跳转的目标地址需要从寄存器索引的操作数中计算出来。

  与上述无条件间接跳转/分支指令的示例同理，但是 RISC-V 架构中没有此类型指令。

对于带条件跳转/分支指令而言，流水线在取指令阶段无法得知该指令的条件是否成立，

因此无法决定是跳还是不跳，理论上指令只有在执行阶段完成之后，才能够解析出最终的跳转结果。假设处理器将取指暂停，一直等到执行阶段完成才继续取指，则会造成大量的流水线空泡周期，从而影响性能。

为了提高性能，现代处理器的取指单元一般会采用分支预测（Branch Prediction）技术。通俗来讲，分支预测需要解决两个方面的问题。

- 预测分支指令是否真的需要跳转？简称为预测“方向”。
- 如果跳转，跳转的目标地址是什么？简称为预测“地址”。

取指时使用预测出的“方向”和“地址”进行取指令的行为称为一种预测取指（Speculative Fetch）的行为，对预取的指令进行执行也便称为一种预测执行（Speculative Execution）的行为。处理器的微架构经过几十年的发展，已经形成了非常成熟的分支预测硬件实现方法，下面予以简介。

2．预测方向

对于“方向”的预测，可以分为静态预测和动态预测两种。

（1）静态预测是最简单的“方向”预测方法，其不依赖于任何曾经执行过的指令信息和历史信息，而是仅依靠这条分支指令本身的信息进行预测。

- 最简单的静态预测方法是总预测分支指令不会发生跳转，因此取指单元便总是顺序取分支指令的下一条指令。待执行阶段之后如果发现需要跳转，则会冲刷流水线（Flush Pipeline）重新进行取指。有关冲刷流水线的介绍，请参见第 9.1.1 节。早期的处理器流水线（以 MIPS 的 5 级流水线为例）往往在第一级取指，然后在第二级译码并对分支真正结果进行判断，因此冲刷流水线后重新取指令需要两个周期。
  为了弥补冲刷流水线造成的性能损失，很多早期的 RISC 架构均使用了“分支延迟槽（Delay Slot）”。最具有代表性的是 MIPS 架构，在很多经典的计算机体系结构教材中均使用 MIPS 对分支延迟槽进行过介绍。分支延迟槽是指在每一条分支指令后面紧跟的一条或者若干条指令不受分支跳转的影响，不管分支是否跳转，后面的几条指令都一定会被执行。分支指令后面的几条指令所在的位置便称为分支延迟槽。由于分支延迟槽中的指令永远被执行而不用被丢弃重取，因此其不会受到冲刷流水线的影响。
- 另一种常见的静态预测方法是 BTFN 预测（Back Taken，Forward Not Taken），即对于向后的跳转预测为跳，向前的跳转则预测为不跳。向后的跳转是指跳转的目标地址（PC 值）比当前分支指令的 PC 值要小。这种 BTFN 方法的依据是在实际的汇编程序中向后分支跳转的情形要多于向前跳转的情形，譬如常见的 for 循环生成的汇编指令往往使用向后跳转的分支指令。

（2）动态预测是指依赖已经执行过的指令的历史信息和分支指令本身的信息综合进行“方向”预测。

- 最简单的分支"方向"动态预测器为"一比特饱和计数器（1-bit Saturating Counter）"，每次分支指令执行之后，便使用此计数器记录上次的"方向"。其预测机制是：下一次分支指令永远采用上次记录的"方向"作为本次的预测。这种预测器结构最简单，但是预测精度不如"两比特饱和计数器（2-bit Saturating Counter）"。
- "两比特饱和计数器（2-bit saturating counter）"是最常见的分支"方向"动态预测器，每次分支指令执行之后，其对应的状态机转换如图 7-2 所示，其预测机制如下。

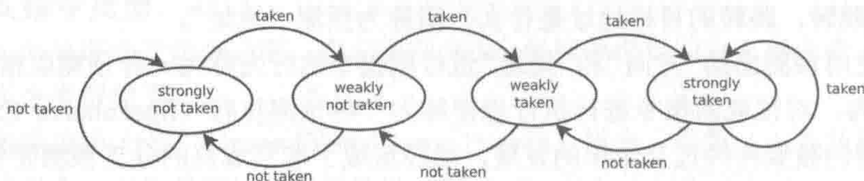


图 7-2　两比特饱和计数器的预测状态机

  当目前状态为"强不需要跳转（strongly not taken）"或者"弱不需要跳转（weakly not taken）"时，预测该指令的方向为"不需要跳转（not taken）"；当目前状态为"弱需要跳转（weakly taken）"或者"强需要跳转（strongly taken）"时，预测该指令的方向为"需要跳转（taken）"。

  每次预测出错之后便会向着相反的方向更改状态机状态，譬如当前状态为"强需要跳转"，会预测为需要跳转，但是实际结果是不需要跳转，则需要将状态机的状态更新为弱不需要跳转。

  由于总共有 4 个状态，譬如从强需要跳转状态需要连续两次预测错误后，才能变到弱不需要跳转，因此具有一定的切换缓冲，其在复杂的程序流中预测精度比简单的"一比特饱和计数器"具有更高的精度。

- "两比特饱和计数器"对于预测一条分支指令很有效，但是处理器执行的指令流中存在着众多的不同分支指令（位于不同的 PC 值位置）。假设只使用一个"两比特饱和计数器"在任何分支指令执行时均进行更新，那么必然会互相冲击，预测的结果会很不理想。最理想的情况是为每一条分支指令都分配专有的"两比特饱和计数器"为其进行预测，但是指令数目众多（32 位架构理论上有 4G 的地址空间），不可能提供巨量的两比特饱和计数器（硬件资源开销无法接受）。所以只能够使用有限个"两比特饱和计数器"组织成一个表格，然后对于每条分支指令使用某种形式寻址方式索引表格中的某个表项的"两比特饱和计数器"。由于表格中的表项数目有限而指令数目众多，因此很多不同的分支指令都会不可避免地指向同样的表项，这种问题称为别名重合（Aliasing）。

目前一般使用各种不同的动态分支预测算法，通俗地讲就是通过采用不同的表格组织方式（控制表格的大小）和索引方式（控制别名重合问题），来提供更高的预测精准率。常见的算法简述如下。

一级预测器

- 最简单的方式是直接将有限个“两比特饱和计数器”组织成一维的表格，称为预测器表格（Predictor Table），并直接使用 PC 值的一部分进行索引。譬如使用 PC 的后 10 位作为索引，则仅需要维护 1000 个表项的表格。
- 这种方法称为一级预测器，所谓“一级”是指其索引仅仅采用指令本身的 PC 值。
- 该方法虽然简单易行，但是索引机制过于简单，很多不同的分支指令都会指向同样的表项（譬如低 10 位相同但是高位不相同的 PC）。并且由于没有考虑到分支指令的上下文执行历史，分支预测的精度不如两级预测器。

两级预测器

- 两级预测器也称为相关预测器（Correlation-Based Branch Predictor）。对于每条分支指令而言，将有限个“两比特饱和计数器”组织成 PHT（Pattern History Table）。使用该分支跳转的历史（Branch History）作为 PHT 的索引。如图 7-3 所示，假设用 n 个比特记录其历史（1 表示 taken，0 表示 not taken），则可以索引 2 的 n 次方个表项。
- 分支历史（Branch History）又可以分为局部历史（Local History）和全局历史（Global History）。局部历史是指每个分支指令自己的分支跳转历史，而全局历史是指所有分支指令的分支跳转历史。

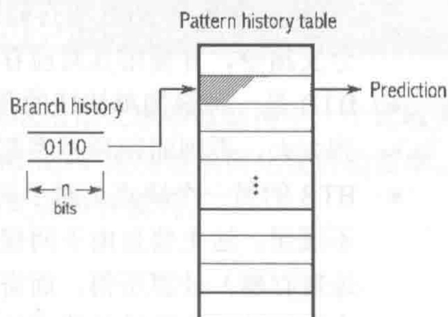


图 7-3 使用分支历史索引 PHT

- 局部分支预测器（Local Branch Predictor）会使用分立的局部历史缓存（Local History Buffer）来保存不同指令的分支历史，每个局部历史缓存有自己对应的 PHT。对于每条分支指令而言，先索引到其对应的局部历史缓存，然后使用局部历史缓存中的历史值索引其对应的 PHT。
- 全局分支预测器（Global Branch Predictor）则仅使用所有分支指令共享的全局历史缓存（Shared Global History Buffer）。全局分支预测器的一个很明显的弊端是它无法区分单独每个分支指令的历史，不同的指令会互相冲击，但是它的优势是比较节省资源。因此全局分支预测器只有在将 PHT 容量做到非常大时，才能体现出其优势，PHT 容量越大，其优势越明显。

  最有代表性的全局分支预测算法是 Gshare 和 Gselect。

  GShare 是 Scott Mcfarling 于 1993 年提出的一种动态分支预测机制，在很多现代的处理器中都有采用。Gshare 算法将分支指令 PC 值的一部分和共享的全局历史缓存进行“异或”运算，然后使用运算的结果作为 PHT 的索引。

  Gselect 算法将分支指令 PC 值的一部分和共享的全局历史缓存直接进行“拼接”运

算，然后使用运算的结果作为 PHT 的索引。

3. 预测地址

对于第 7.1.4 节中介绍的直接跳转/分支（Direct Jump/Branch）指令，分支目标地址需要使用当前的 PC 值和对应取回的指令字中的立即数进行加法运算；而对于间接跳转/分支（Indirect Jump/Branch）指令，由于分支目标地址需要使用寄存器索引的操作数（基地址寄存器）和指令字中的立即数进行加法运算，只能在流水线的执行阶段才计算出分支目标地址。在现代高速的处理器中，这些都是不可能在一个周期内完成的，在高速的处理器中连续取下一条指令之前，甚至连译码判断当前取到的指令是否属于分支指令都无法及时在一个周期内完成。

因此为了能够连续不断地取指，需要预测分支的目标“地址”，常见的技术简述如下。

（1）BTB

- 分支目标缓存（Branch Target Buffer，BTB）技术是指使用容量有限的缓存保存最近执行过的分支指令的 PC 值，以及它们的跳转目标地址。对于后续需要取指的每条 PC 值，将其与 BTB 中存储的各个 PC 值进行比较，如果出现匹配，则预测这是一条分支指令，并使用其对应存储的跳转目标地址作为预测的跳转地址。
- BTB 是一种最简单快捷的预测“地址”方法，但是其缺点之一是不能将 BTB 容量做到太大，否则面积和时序都无法接受。
- BTB 的另一个缺点是对于间接跳转/分支（Indirect Jump/Branch）指令的预测效果并不理想。这主要是由于间接跳转/分支的目标地址是使用寄存器索引的操作数（基地址寄存器）计算所得，而寄存器中的值随着程序执行可能每次都不一样，因此 BTB 中存储的上次跳转的目标地址并不一定等于本次跳转的目标值。

（2）RAS

- 返回地址堆栈（Return Address Stack，RAS）技术是指使用容量有限的硬件堆栈（一种“先进后出”的结构）来存储函数调用的返回地址。
- 在 7.1.4 节中介绍过，以 RISC-V 架构为例，间接跳转/分支（Indirect Jump/Branch）可以用于函数的调用和返回。而函数的调用和返回在程序中往往是成对出现的，因此可以在函数调用（使用分支跳转指令）时将当前 PC 值加 4（或者 2）。即其顺序执行的下一条指令的 PC 值压入 RAS 堆栈中，等到函数返回（使用分支跳转指令）时将 RAS 中的值弹出，这样就可以快速地为该函数返回的分支跳转指令预测目标地址。
- 只要程序是在正常执行，其函数的调用和返回成对出现，那么 RAS 便能够提供较高的预测准确率。当然由于 RAS 的深度有限，如果程序中出现很多次函数嵌套，需要不断地压入堆栈，造成堆栈溢出，则会影响到预测准确率，硬件需要特殊处理该情形。

（3）Indirect BTB

- 间接 BTB（Indirect BTB）是指专门为间接跳转/分支（Indirect Jump/Branch）指令而设计的 BTB，它与普通 BTB 类似，存储较多历史目标地址，但是通过高级的索引方

法进行匹配（而不是简单的 PC 值比较），可以说结合了 BTB 和动态两级预测器的技术，能够提供较高跳转目标地址预测成功率。但其缺点是硬件开销非常大，只有在高级的处理器中才会使用。

- 还有其他的技术能够提高间接跳转/分支（Indirect Jump/Branch）指令的跳转目标地址预测成功率，本文在此不做赘述。

4. 其他拓展

本书在此仅对分支预测常见的技术进行了简介。分支预测是处理器微架构中非常重要且比较复杂的内容，若要彻底阐述明晰需要数十页的篇幅。很多处理器体系结构的教材中都有专门的章节详述，本书在此不做赘述。推荐读者阅读维基百科关于分支预测的词条网页（请在维基百科中搜索“Branch_predictor”）。

7.2 RISC-V 架构特点对于取指的简化

上一节讨论了处理器取指的相关背景和技术，可以说取指是处理器微架构中一个比较关键且复杂的部分。在第 2 章中，我们曾总结性地探讨过 RISC-V 架构追求简化硬件的哲学。具体对于取指而言，RISC-V 架构的如下特点可以大幅简化其硬件实现。

- 规整的指令编码格式。
- 指令长度指示码放于低位。
- 简单的分支跳转指令。
- 没有分支延迟槽指令。
- 提供明确的静态分支预测依据。
- 提供明确的 RAS 依据。

下文予以分别论述。

7.2.1 规整的指令编码格式

取指时如果能够尽快译码出当前取出的指令类型（譬如是否属于分支跳转指令），将有利于取指逻辑的效率和实现。在第 2 章中曾经论述过 RISC-V 架构得益于后发优势，并总结了多年来处理器发展的教训，其指令集编码非常规整，可以非常便捷地译码出指令的类型及其使用的操作数寄存器索引（Index）或者立即数，从而简化硬件设计。

7.2.2 指令长度指示码放于低位

为了提高代码密度，RISC-V 定义了一种可选的压缩（Compressed）指令子集，由字母

C表示。如果支持此压缩子集，就会有32位和16位指令混合交织在一起的情形。

为了支持16位指令的取指，取指逻辑每取一条指令之后需要以最快的速度译码判断出当前指令的宽度是16位或者32位。得益于后发优势和多年来处理器发展的教训，RISC-V架构的制定者预先考虑了这个问题。如图7-4所示，所有的RISC-V指令编码的最低几位专门用于编码表示指令的长度。将指令长度指示码放在指令的最低位，可以方便取指逻辑在顺序取指的过程中以最快的速度译码出指令的长度，极大地简化硬件设计。譬如，取指逻辑在仅取到16位指令字时就可以进行译码判断，当前指令是16位长还是32位长，而无需等待另外一半的16位指令字取到之后，才开始译码。

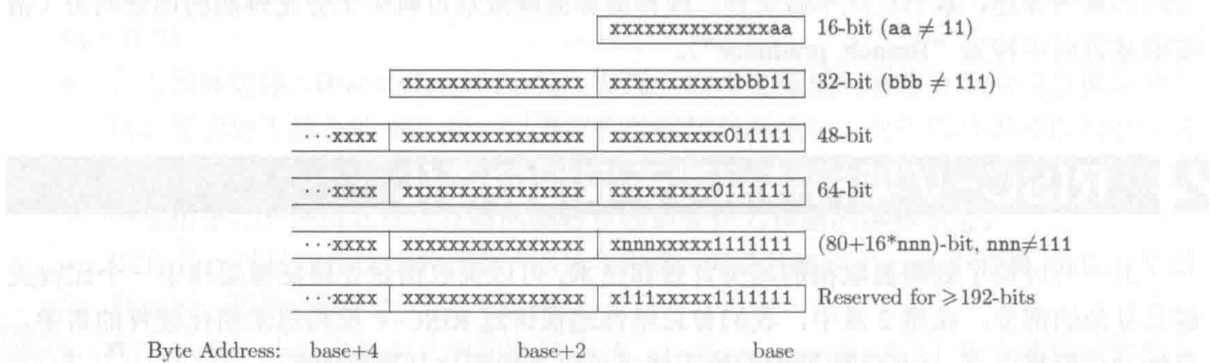


图7-4 RISC-V指令长度的编码信息

另外，由于16位的压缩指令子集是可选的，假设处理器不支持此压缩指令子集而仅支持32位指令，甚至可以将指令字的低2位忽略不存储（因为其肯定固定为11），从而节省6.25%的指令缓存（I-Cache）的开销。

注意：从图7-4中可以看出，RISC-V架构甚至可以支持48位和64位等不同的指令长度，但是这些均属于非必需的罕见指令，本书在此对其不做介绍。

7.2.3 简单的分支跳转指令

RISC-V的基本整数指令子集中的分支跳转指令总结如表7-1所示。

表7-1 RV32I架构中的分支跳转指令

分 组	指 令	描 述
无条件直接跳转/分支指令	jal	• jal（jump and link）指令的汇编示例如“jal x5, offset” • jal指令一定会发生跳转，其使用编码在指令字中的20位立即数（有符号数）作为偏移量（offset）。该偏移量乘以2，然后与当前指令所在的地址相加，生成得到最终的跳转目标地址 • jal指令将下一条指令的PC（当前指令PC+4）的值写入其结果寄存器

续表

分　组	指　令	描　述
无条件间接跳转/分支指令	jalr	• jalr（jump and link-register）指令的汇编示例如“jalr x1, x6, offset” • jalr 指令一定会发生跳转，其使用编码在指令字中的 12 位立即数（有符号数）作为偏移量（offset），与 jalr 的另外一个寄存器索引的操作数（基地址寄存器）相加，得到最终的跳转目标地址 • jalr 指令将下一条指令的 PC（当前指令 PC+4）的值写入其结果寄存器
带条件直接跳转/分支指令	beq	两个整数操作数相等则跳转
	bne	两个整数不相等则跳转
	blt	第一个有符号数小于第二个有符号数则跳转
	bltu	第一个无符号数小于第二个无符号数则跳转
	bge	第一个有符号数大于等于第二个有符号数则跳转
	bgeu	第一个无符号数大于等于第二个无符号数则跳转

如表 7-1 所示，RISC-V 架构有两条无条件跳转指令（Unconditional Jump）——jal 与 jalr 指令。jal 指令可以用于进行子程序调用，同时将子程序返回地址存在 jal 指令的目标结果寄存器（链接寄存器，Link Register）中。jalr 指令可以用于子程序返回指令，通过将 jal 指令（跳转进入子程序）保存的链接寄存器用于 jalr 指令的基地址寄存器，则可以从子程序返回。

如表 7-1 所示，RISC-V 架构有 6 条带条件分支指令（Conditional Branch），这种带条件的分支指令跟普通的运算指令一样，直接使用 2 个整数操作数，然后对其进行比较。如果比较的条件满足时则进行跳转，因此这是将比较与跳转两个操作放到了一个指令中完成。这种带条件分支指令使用 12 位的有符号数作为偏移量。该偏移量乘以 2 后与当前指令所在的地址相加，生成得到最终的跳转目标地址。

对于类似带条件分支功能，很多的其他 RISC 架构的处理器需要使用两条独立的指令。第一条指令先使用比较指令（Compare），比较的结果被保存到状态寄存器中。第二条指令使用跳转指令，判断前一条指令保存在状态寄存器当中的比较结果为真时，则进行跳转。相比而言，RISC-V 架构将比较与跳转两个操作放到了一个指令的方式不仅减少了指令的条数，而且在硬件设计上更加简单。

RISC-V 架构中 16 位压缩指令子集也定义了若干分支跳转指令，总结如表 7-2 所示。但是第 2.2.11 节曾经提到，RISC-V 架构的一个精妙之处在于其 16 位的指令一定能够对应到一条 32 位的等效指令，分支跳转指令也不例外，因此功能与基本整数指令子集中的分支跳转指令一致，在此不再赘述。

表 7-2　　RV32C 架构中的分支跳转指令

分　组	指　令	等效的 RV32I 指令
无条件直接跳转/分支指令	C.J	jal x0, offset[11:1]

续表

分　组	指　令	等效的 RV32I 指令
无条件直接跳转/分支指令	C.JAL	jal x1, offset[11:1] 注意：由于 C.JAL 的指令宽度是 16 位，因此下一条指令的 PC 值为当前 PC+2
无条件间接跳转/分支指令	C.JR	jalr x0, rs1, 0
	C.JALR	jalr x1, rs1, 0 注意：由于 C.JAL 的指令宽度是 16 位，因此下一条指令的 PC 值为当前 PC+2
带条件直接跳转/分支指令	C.BEQZ	beq rs1, x0, offset[8:1]
	C.BNEZ	bne rs1, x0, offset[8:1]

7.2.4 没有分支延迟槽指令

分支延迟槽是指在每一条分支指令后面紧跟的一条或者若干条指令不受分支跳转的影响，不管分支是否跳转，这后面的几条指令都一定会被执行。分支指令后面的几条指令所在的位置便称为分支延迟槽。由于分支延迟槽中的指令永远被执行而不用被丢弃重取，使得其不会受到流水线冲刷（Pipeline Flush）的影响，也降低了对分支预测精度的要求。很多早期的 RISC 架构均使用了分支延迟槽的技术，最具有代表性的便是 MIPS 架构，在很多经典的计算机体系结构教材中均使用 MIPS 对分支延迟槽有所介绍。

分支延迟槽在早期的 RISC 架构中被采用，主要是因为早期的 RISC 处理器流水线比较简单，没有使用高级的硬件动态分支预测器，使用分支延迟槽能够取得可观的效果。然而这种分支延迟槽使得处理器的硬件设计变得极为别扭，尤其是对于取指部分的硬件设计将会比较烦琐。

RISC-V 架构放弃了分支延迟槽，RISC-V 架构的制定者认为放弃分支延迟槽的得大于失。因为现代的高性能处理器的分支预测算法精度已经非常高，可以有强大的分支预测电路，保证处理器能够准确地预测跳转执行达到高性能。而对于低功耗小面积的处理器可以选择非常简单的电路进行实现，由于无须支持分支延迟槽所带来的硬件极大简化也能进一步减少功耗和提高时序。

7.2.5 提供明确的静态分支预测依据

静态分支预测是一种最简单的预测技术，但是静态分支预测往往固定地预测向后跳转（或者向前跳转）为需要跳转（Taken）。如果软件实际执行中未必如此，则会造成预测失败。

RISC-V 架构中明确规定，编译器生成的代码应该尽量优化，使得向后跳转的分支指令比向前跳转的分支指令有更大的概率进行跳转。如此，对于使用静态预测的低端处理器，可以保证其行为和软件行为匹配，最大化地提高静态预测的准确率。

7.2.6 提供明确的 RAS 依据

RAS（Return Address Stack）可以用于函数返回地址的预测，这是目前处理器设计的常用技术。但是 RAS 需要能够明确地判定什么指令属于“函数调用”类型的分支跳转指令从而进行压栈，什么指令属于“函数返回”类型的分支跳转指令从而进行出栈。

RISC-V 架构中明确规定，如果使用 jal 指令且目标寄存器索引值 rd 等于 x1 或者 x5，则属于需要进行 RAS 压栈；如果使用 jalr 指令，则按照使用的寄存器值（rs1 和 rd）的不同，明确规定了相应的 RAS 压栈或者出栈行为，如图 7-5 所示（注意：图中的 link 表示 x1 或者 x5）。

rd	*rs1*	*rs1=rd*	RAS action
!link	*!link*	-	none
!link	*link*	-	pop
link	*!link*	-	push
link	*link*	0	push and pop
link	*link*	1	push

图 7-5　RISC-V 架构中对 jalr 指令规定的 RAS 行为

通过在架构文档中明确规定，并规定软件编译器必须按照此原则生成汇编代码，能够保证硬件的行为和软件匹配吻合，从而最大化地提高 RAS 的预测准确性。

7.3 蜂鸟 E200 处理器的取指实现

蜂鸟 E200 处理器核的取指子系统处于流水线中的位置如图 7-6 中的圆圈区域所示，主要包括取指令单元（Instruction Fetch Unit，IFU）和 ITCM。

图 7-6　蜂鸟 E200 处理器核的取指子系统结构示意图

7.3.1 IFU 总体设计思路

蜂鸟 E200 的 IFU 微架构如图 7-7 所示，其主要包括如下功能。

- 对取回的地址进行简单译码（Mini-Decode）。
- 简单的分支预测（Simple-BPU）。
- 生成取指的 PC（PC 生成）。
- 根据 PC 的地址访问 ITCM 或 BIU（地址判断和 ICB 总线控制）。

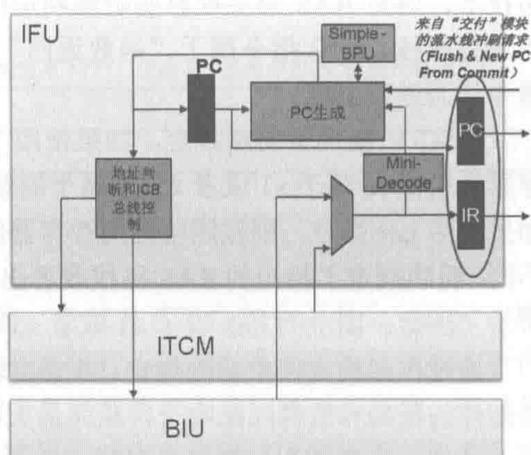


图 7-7　蜂鸟 E200 处理器核 IFU 微架构示意图

IFU 在取出指令后，会将其放置于和 EXU 单元接口的 IR（Instruction Register）寄存器中。该指令的 PC 值也会被放置于和 EXU 单元接口的 PC 寄存器中，如图 7-7 中的圆圈所示，EXU 单元将使用此 IR 和 PC 进行后续的执行操作。有关 EXU 单元的实现细节，请参见第 8 章。

在第 7.1.1 节中曾经探讨，取指令的要点是“快”和“连续不断”。

（1）针对“快”，蜂鸟 E200 的设计理念如下。

- 蜂鸟 E200 假定绝大多数的取指都发生在 ITCM 中，这种假定具有合理性。因为蜂鸟 E200 是面向嵌入式超低功耗场景设计的小面积处理器，没有使用 I-Cache，主要使用 ITCM 进行执行的存储以满足实时性的要求。这种级别的嵌入式处理器核的程序代码量不大，往往可以全部加载在 ITCM 中执行。
- 蜂鸟 E200 的 ITCM 使用单周期访问的 SRAM，即可以一个周期就从 ITCM 中取回一条指令。因此假设指令存放于 ITCM 中，从 ITCM 中取指理论上可以做到“快”。
- 对于某些特殊情况，指令需要从外部存储器中读取（譬如系统上电后的引导程序可能需要从外部 Flash 中读取）。此时，IFU 需要通过 BIU 使用系统存储接口访问外部的存储器，访问延迟不可能做到单周期访问。因此对于外部存储器的取指，性能无法做到“快”。但是如上文所述，蜂鸟 E200 假定绝大多数的取指都发生在 ITCM 中，对于这种外部存储器的访问出现的情形非常少，因此对这种情况不做优化。
- 运行于蜂鸟 E200 上的软件也应该尽量利用“绝大多数的取指都发生在 ITCM 中”的假定，尽可能发挥处理器核的性能。

（2）针对“连续不断”，蜂鸟 E200 的设计思路如下。

- 为了能够连续不断地取指令，需要每个周期都能生成下一条待取指令的 PC 值，因此需要判别本指令的类型是普通指令还是分支跳转指令，从而理论上需要对当前取回的指令进行译码。

- 蜂鸟 E200 的 IFU 选择直接将取回的指令在同一个周期内进行部分译码（如图 7-7 中的 Mini-Decode）。如果译码的信息指示当前指令为分支跳转指令，则 IFU 直接在同一个周期内进行分支预测（如图 7-7 中的 Simple-BPU）。最后，使用译码得出的信息和分支预测的信息进行下一条待取指令 PC 的生成（如图 7-7 中的 PC 生成）。
- 由于在一个周期内完成了指令读取（假设是从 ITCM 中取指）、部分译码、分支预测和生成下一条待取指令的 PC 等连贯操作，因此理论上可以做到“连续不断”。
- 当然，由于在一个周期内完成了上述众多步骤，时序上的关键路径可能会制约蜂鸟 E200 能达到的最高主频。一方面得益于第 7.2 节所述的 RISC-V 架构的简单性，指令的部分译码和分支预测消耗的逻辑延迟并不算太大；另一方面蜂鸟 E200 的设计理念重在强调超低功耗和小面积，对于最高主频选择适当的放弃。

取指令需要使用到分支预测技术，针对“分支预测”，蜂鸟 E200 的设计理念如下。

- 蜂鸟 E200 作为一款面向超低功耗的处理器，分支预测采用最简单的静态预测。
- 得益于 RISC-V 架构明确提供了对于静态预测的依据，因此蜂鸟 E200 的静态预测对于向后跳转的条件分支指令预测为真的跳转，而对于向前跳转的条件分支指令预测为不需要跳转。

下文对 IFU 的不同子模块予以分别论述。

7.3.2 Mini-Decode

Mini-Decode 模块主要用于对取回的指令进行译码，要点如下。

- Mini-Decode 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                              // 存放RTL的目录
         |----e203                        // E203核和SoC的RTL目录
              |----core                   // 存放e203 Core的RTL代码
                   |----e203_ifu_minidec.v    // Mini-Decode模块
```

- 之所以称为 Mini-Decode，是因为此处的译码并不需要完整译出指令的所有信息，而只需要译出 IFU 所需的部分指令信息。包括此指令是属于普通指令还是分支跳转指令、分支跳转指令的类型和细节。
- Mini-Decode 模块内部也是例化调用一个完整的 Decode 模块，但是将其不相关的输入信号接零、输出信号悬空不连接，从而使得综合工具将完整 Decode 模块中无关逻辑优化掉，成为一个 Mini-Decode。之所以使用这种方式，是因为我们只想维护一份 Decode 模块的源代码，而不是分别写一个 Full-Decode 和 Mini-Decode 模块，从而避免两头维护同一个模块而出错的情形（在工程中，修改了一份文件而忽略了另外一

份文件造成功能出错的情形时有发生）。

- Mini-Decode 模块的相关的源代码片段如下所示。

```
// e203_ifu_minidec.v 源代码片段
module e203_ifu_minidec(
  //////////////////////////////////////////////////////////////
  // The IR stage to Decoder
  input ['E203_INSTR_SIZE-1:0] instr, // 取回的指令输入进行部分译码

  //////////////////////////////////////////////////////////////
  // The Decoded Info-Bus

  output dec_rs1en,
  output dec_rs2en,
  output ['E203_RFIDX_WIDTH-1:0] dec_rs1idx,
  output ['E203_RFIDX_WIDTH-1:0] dec_rs2idx,
......
  output dec_rv32, //指示当前指令为16位还是32位
  output dec_bjp,  //指示当前指令属于普通指令还是分支跳转指令
  output dec_jal,  // 属于JAL指令
  output dec_jalr, // 属于JALR指令
  output dec_bxx,  // 属于Bxx指令（BEQ，BNE等带条件分支指令）
  output ['E203_RFIDX_WIDTH-1:0] dec_jalr_rs1idx,
  output ['E203_XLEN-1:0] dec_bjp_imm

  );

 // 此模块内部例化调用一个完整的Decode模块，但是将其不相关的输入信号接零、输出信号悬空不连
接，从而使得综合工具将完整Decode模块中无关逻辑优化掉，成为一个Mini-Decode
  e203_exu_decode u_e203_exu_decode(

  .i_instr(instr),
  .i_pc('E203_PC_SIZE'b0),//不相关的输入信号接零
  .i_prdt_taken(1'b0),
  .i_muldiv_b2b(1'b0),

  .i_misalgn  (1'b0),
  .i_buserr   (1'b0),

  .dbg_mode   (1'b0),

  .dec_misalgn(),//不相关的输出信号悬空不连接
  .dec_buserr(),
  .dec_ilegl(),

  .dec_rs1x0(),
```

```
    .dec_rs2x0(),
    .dec_rs1en(dec_rs1en),
    .dec_rs2en(dec_rs2en),
    .dec_rdwen(),
    .dec_rs1idx(dec_rs1idx),
    .dec_rs2idx(dec_rs2idx),
    .dec_rdidx(),
    .dec_info(),
    .dec_imm(),
    .dec_pc(),


    .dec_mulhsu(dec_mulhsu),
    .dec_mul    (dec_mul    ),
    .dec_div    (dec_div    ),
    .dec_rem    (dec_rem    ),
    .dec_divu   (dec_divu   ),
    .dec_remu   (dec_remu   ),

    .dec_rv32(dec_rv32),
    .dec_bjp (dec_bjp ),
    .dec_jal (dec_jal ),
    .dec_jalr(dec_jalr),
    .dec_bxx (dec_bxx ),

    .dec_jalr_rs1idx(dec_jalr_rs1idx),
    .dec_bjp_imm    (dec_bjp_imm    )
  );

endmodule
```

7.3.3 Simple-BPU 分支预测

Simple-BPU 模块主要用于对取回的指令进行 Mini-Decode 后发现的分支跳转指令进行分支预测。之所以称为 Simple-BPU，是由于蜂鸟 E200 作为一款面向超低功耗的处理器，只采用了最简单的静态预测，并未采用其他高级动态预测技术。Simple-BPU 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
     |----rtl                          // 存放 RTL 的目录
          |----e203                    // E203 核和 SoC 的 RTL 目录
               |----core               // 存放 e203 Core 的 RTL 代码
                    |----e203_ifu_litebpu.v       // Simple-BPU 模块
```

1. 带条件直接跳转指令

对于带条件直接跳转指令 Bxx 指令（BEQ、BNE 等指令），使用静态预测（向后跳转则预

测为需要跳，否则预测为不需要跳）。而对于其跳转目标地址，Simple-BPU 按照指令的定义，使用其 PC 和立即数表示的 offset 相加得到其跳转目标地址。相关的源代码片段如下所示。

```
// e203_ifu_litebpu.v 源代码片段

......
// 如果立即数表示的偏移量（offset）为负数（最高位符号位为 1），意味着方向为向后跳转，预测为需要跳转。
   assign prdt_taken   = (dec_jal | dec_jalr | (dec_bxx & dec_bjp_imm['E203_XLEN-1]));

// 由于 PC 计算需要使用到加法器，为了节省面积，所有的 PC 计算均共享同一个加法器。
// 此处生成分支预测器进行 PC 计算所需的操作数，将送给共享的加法器进行计算。
// 生成加法器的操作数一：如果是 Bxx 指令，便使用它本身的 PC
  assign prdt_pc_add_op1 = (dec_bxx | dec_jal) ? pc['E203_PC_SIZE-1:0]
                         : (dec_jalr & dec_jalr_rs1x0) ? 'E203_PC_SIZE'b0
                         : (dec_jalr & dec_jalr_rs1x1) ? rf2bpu_x1['E203_PC_SIZE-1:0]
                         : rf2bpu_rs1['E203_PC_SIZE-1:0];

// 生成加法器的操作数二：使用立即数表示的偏移量
  assign prdt_pc_add_op2 = dec_bjp_imm['E203_PC_SIZE-1:0];

......
```

2. 无条件直接跳转指令 jal

对于无条件直接跳转指令 jal，由于其一定会跳转，因此无须预测其跳转方向。而对于其跳转目标地址，Simple-BPU 按照指令的定义，使用其 PC 和立即数表示的 offset 相加得到其跳转目标地址。相关的源代码片段如下所示。

```
// e203_ifu_litebpu.v 源代码片段

......

// 由于 PC 计算需要使用到加法器，为了节省面积，所有的 PC 计算均共享同一个加法器。
// 此处生成分支预测器进行 PC 计算所需的操作数，将送给共享的加法器进行计算。

// 生成加法器的操作数一：如果是 jal 指令，便使用它本身的 PC
  assign prdt_pc_add_op1 = (dec_bxx | dec_jal) ? pc['E203_PC_SIZE-1:0]
                         : (dec_jalr & dec_jalr_rs1x0) ? 'E203_PC_SIZE'b0
                         : (dec_jalr & dec_jalr_rs1x1) ? rf2bpu_x1['E203_PC_SIZE-1:0]
                         : rf2bpu_rs1['E203_PC_SIZE-1:0];

// 生成加法器的操作数二：使用立即数表示的偏移量
  assign prdt_pc_add_op2 = dec_bjp_imm['E203_PC_SIZE-1:0];

......
```

3. 无条件间接跳转指令 jalr

对于无条件间接跳转指令 jalr，由于其一定会跳转，因此无须预测其跳转方向。而对于

跳转目标地址，jalr 的跳转目标计算所需的基地址来自于其 rs1 索引的操作数，需要从通用寄存器组（Regfile）中读取，并且还可能和正在 EXU 执行的指令形成 RAW 数据相关性。蜂鸟 E200 采用了一种比较巧妙的方案，根据 rs1 的索引值不同而采取不同的方案，要点如下。

- 如果 rs1 的索引号是 x0，则意味着直接使用常数 0（根据 RISC-V 架构定义 x0 表示常数 0），无须从 Regfile 中读取。相关的源代码片段如下所示。

```
// e203_ifu_litebpu.v 源代码片段

......

//判定 rs1 的索引号是 x0
  wire dec_jalr_rs1x0 = (dec_jalr_rs1idx == 'E203_RFIDX_WIDTH'd0);
    ......

// 由于 PC 计算需要使用到加法器，为了节省面积，所有的 PC 计算均共享同一个加法器。
// 此处生成分支预测器进行 PC 计算所需的操作数，将送给共享的加法器进行计算。

// 生成加法器的操作数一：
 assign prdt_pc_add_op1 = (dec_bxx | dec_jal) ? pc['E203_PC_SIZE-1:0]
              //如果是 JALR 指令且 rs1 为 x0，便使用常数 0
                    : (dec_jalr & dec_jalr_rs1x0) ? 'E203_PC_SIZE'b0
                    : (dec_jalr & dec_jalr_rs1x1) ? rf2bpu_x1['E203_PC_SIZE-1:0]
                    : rf2bpu_rs1['E203_PC_SIZE-1:0];

// 生成加法器的操作数二：使用立即数表示的偏移量
 assign prdt_pc_add_op2 = dec_bjp_imm['E203_PC_SIZE-1:0];

......
```

- 如果 rs1 的索引号是 x1，由于 x1 常用于 link 寄存器作为函数返回跳转指令，因此蜂鸟 E200 对其进行特别加速，将 x1 从处于 EXU 的 Regfile 中直接拉线取出（不需要占用 Regfile 的读端口）。为了防止正在处于 EXU 中执行的指令需要写回 x1 造成 RAW 数据相关性，Simple-BPU 需要判定当前的 EXU 指令没有写回 x1，并且还需要判定 OITF（有关 OITF 的相关信息请参见第 8.3.7 节）为空。相关的源代码片段如下所示。

```
// e203_ifu_litebpu.v 源代码片段

......
//判定 rs1 的索引号是 x1
  wire dec_jalr_rs1x1 = (dec_jalr_rs1idx == 'E203_RFIDX_WIDTH'd1);
......
//判定 x1 是否可能与 EXU 中的指令存在潜在的 RAW 数据相关性。在两种情况下可能出现 RAW 相关性:
//        1:  OITF 不为空，意味着可能有长指令正在执行，其结果可能会写回 x1。当然也有可能
//            长指令写回的结果寄存器不是 x1，但是此处我们采取简单的保守估计，对于造成
//            的性能损失不在意。有关 OITF 和长指令的相关信息，请参见第 8.3.7 节了解更多
//            信息。
//        2:  处于 IR 寄存器中的指令的写回目标寄存器的索引号为 x1，意味着有 RAW 数据相关性。
```

```
  wire jalr_rs1x1_dep = dec_i_valid & dec_jalr & dec_jalr_rs1x1 & ((~oitf_e
mpty) | (jalr_rs1idx_cam_irrdidx));

//如果存在着 x1 的 RAW 相关性，则将 bpu_wait 拉高，此信号将阻止 IFU 生成下一个 PC，等待相关
//性解除。
//因此就性能而言，
//     1：如果 x1 依赖于 EXU 的 ALU 指令（大多数情况），需要等待 1 个周期 ALU 执行完毕写回 Regfile
//        后，bpu_wait 信号才会拉低进而继续取指。流水线中会因此出现 1 个周期的空泡性能损失。
//     2：如果 x1 和 EXU 中的指令没有数据相关性，则不会造成将 bpu_wait 拉高，不会有任何的性能损失。
  assign bpu_wait = jalr_rs1x1_dep | jalr_rs1xn_dep | rs1xn_rdrf_set;

// 由于 PC 计算需要使用到加法器，为了节省面积，所有的 PC 计算均共享同一个加法器。
// 此处生成分支预测器进行 PC 计算所需的操作数，将送给共享的加法器进行计算。

// 生成加法器的操作数一：
  assign prdt_pc_add_op1 = (dec_bxx | dec_jal) ? pc['E203_PC_SIZE-1:0]
                       : (dec_jalr & dec_jalr_rs1x0) ? 'E203_PC_SIZE'b0
                  //如果是 JALR 指令且 rs1 为 x1，便使用从 Regfile 中硬连线出来的 x1 值
                       : (dec_jalr & dec_jalr_rs1x1) ? rf2bpu_x1['E203_PC_SIZE-1:0]
                       : rf2bpu_rs1['E203_PC_SIZE-1:0];

// 生成加法器的操作数二：使用立即数表示的偏移量
  assign prdt_pc_add_op2 = dec_bjp_imm['E203_PC_SIZE-1:0];

......
```

- 如果 rs1 的索引号是除了 x0 和 x1 的其他寄存器（简称 xn），蜂鸟 E200 对其不进行特别加速。xn 需要使用 Regfile 的第 1 个读端口（Read Port 1）从 Regfile 中读取出来，因此需要判定当前第 1 个读端口是否空闲且不存在资源冲突（参见第 8.3.4 节了解有关 Regfile 实现的信息）。并且，为了防止正在处于 EXU 中执行的指令需要写回 xn 造成 RAW 数据相关性，Simple-BPU 需要判定当前的 EXU 中没有任何指令。相关的源代码片段如下所示。

```
// e203_ifu_litebpu.v 源代码片段

......
//判定 rs1 的索引号是 xn
  wire dec_jalr_rs1xn = (~dec_jalr_rs1x0) & (~dec_jalr_rs1x1);
......
//判定 xn 是否可能与 EXU 中的指令存在潜在的 RAW 数据相关性，在两种情况下可能出现 RAW 相关性。
//          1:  OITF 不为空，意味着可能有长指令正在执行，其结果可能会写回 xn。当然也有可能
//              长指令写回的结果寄存器不是 xn，但是此处我们采取简单的保守估计，对于造成
//              的性能损失不在意。有关 OITF 和长指令的相关信息，请参见第 8.3.7 节
//          2:  在 IR 寄存器中存在指令，意味着可能会写回 xn。当然也有可能该指令写回的结果寄
//              存器不是 xn，但是此处我们采取简单的保守估计，不在意造成的性能损失。
```

```
    wire jalr_rs1xn_dep = dec_i_valid & dec_jalr & dec_jalr_rs1xn & ((~oitf_emp
ty) | (~ir_empty));

    ......

    //需要征用 Regfile 的第 1 个读端口从 Regfile 中读取 xn 的值，需要判断第 1 个读端口是否空
    //闲不存在资源冲突。
    //如果没有资源冲突和数据冲突时，则将征用第 1 个读端口的使能置高
      wire rs1xn_rdrf_set = (~rs1xn_rdrf_r) & dec_i_valid & dec_jalr & dec_jalr
_rs1xn & ((~jalr_rs1xn_dep) | jalr_rs1xn_dep_ir_clr);
      wire rs1xn_rdrf_clr = rs1xn_rdrf_r;
      wire rs1xn_rdrf_ena = rs1xn_rdrf_set |    rs1xn_rdrf_clr;
      wire rs1xn_rdrf_nxt = rs1xn_rdrf_set | (~rs1xn_rdrf_clr);

      sirv_gnrl_dfflr #(1) rs1xn_rdrf_dfflrs(rs1xn_rdrf_ena, rs1xn_rdrf_nxt, rs
1xn_rdrf_r, clk, rst_n);

    //生成征用第 1 个读端口的使能信号，该信号将加载和 IR 寄存器位于同一级的 rs1 索引(index)寄存器，
    //从而读取 Regfile
      assign bpu2rf_rs1_ena = rs1xn_rdrf_set;

    //如果存在着 xn 的 RAW 相关性，则将 bpu_wait 拉高，不仅如此，在征用第 1 个读端口的周期也会
    // 将 bpu_wait 拉高。此信号将阻止 IFU 生成下一个 PC，直到相关性解除并且从 Regfile 中已经读
    // 出 xn 的值。
    //因此就性能而言，由于需要征用 Regfile 的第 1 个读端口读取 xn 的值，即便没有数据相关性，
    // 也需要最少等待 1 个周期。
     assign bpu_wait = jalr_rs1x1_dep | jalr_rs1xn_dep | rs1xn_rdrf_set;

    // 由于 PC 计算需要使用到加法器，为了节省面积，所有的 PC 计算均共享同一个加法器。
    // 此处生成分支预测器进行 PC 计算所需的操作数，将送给共享的加法器进行计算。

    // 生成加法器的操作数一：
      assign prdt_pc_add_op1 = (dec_bxx | dec_jal) ? pc['E203_PC_SIZE-1:0]
                         : (dec_jalr & dec_jalr_rs1x0) ? 'E203_PC_SIZE'b0
                         : (dec_jalr & dec_jalr_rs1x1) ? rf2bpu_x1['E203_PC_SIZE-1:0]
                  //如果是 JALR 指令且 rs1 为 xn，便使用从 Regfile 的第 1 个读端口中读取出来
                  // 的 xn 值
                         : rf2bpu_rs1['E203_PC_SIZE-1:0];

    // 生成加法器的操作数二：使用立即数表示的偏移量
      assign prdt_pc_add_op2 = dec_bjp_imm['E203_PC_SIZE-1:0];

    ......
```

7.3.4 PC 生成

PC 生成逻辑用于产生下一个待取指令的 PC，PC 生成根据不同的情形需要不同处理。

- 如果是 reset 后的第一次取指，使用蜂鸟 E200 的 CPU-TOP 顶层输入信号 pc_rtvec 指示的值作为第一次取指的 PC 值。用户可以通过在 SoC 顶层集成时，将此信号赋予不同的值来控制 PC 的复位默认值。
- 对于顺序取指的情形，根据当前指令是 16 位指令还是 32 位指令判断自增值。如果是 16 位指令，顺序取指的下一条指令的 PC 为 PC+2；如果是 32 位指令，则顺序取指的下一条指令的 PC 为 PC+4。
- 如果是分支指令，则使用 Simple-BPU 预测的跳转目标地址。
- 如果是来自于 EXU 的流水线冲刷，则使用 EXU 送过来的新 PC 值。

PC 生成的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                // 存放 RTL 的目录
        |----e203                           // E203 核和 SoC 的 RTL 目录
            |----core                       // 存放 e203 Core 的 RTL 代码
                |----e203_ifu_ifetch.v              // 包含 PC 生成的 fetch 模块
```

PC 生成的相关源代码片段如下所示。

```
    // e203_ifu_ifetch.v 源代码片段

//生成 PC 自增值：如果当前指令为 32 位，则顺序取指的下一条指令 PC 需要加 4，否则加 2
  wire [2:0] pc_incr_ofst = minidec_rv32 ? 3'd4 : 3'd2;

  wire ['E203_PC_SIZE-1:0] pc_nxt_pre;
  wire ['E203_PC_SIZE-1:0] pc_nxt;

//表示跳转取指令的信号：如果是分支跳转指令，且 Simple-BPU 预测需要跳转，则跳转取指
  wire bjp_req = minidec_bjp & prdt_taken;

// 由于 PC 计算需要使用到加法器，为了节省面积，所有的 PC 计算均共享同一个加法器。
// 此处选择加法器的输入。

  wire ['E203_PC_SIZE-1:0] pc_add_op1 =
                          // 如果是跳转取指，则使用 Simple-BPU 产生的加法操作数一
                             bjp_req ? prdt_pc_add_op1     :
                          // 如果是 reset 后取指，则使用 pc_rtvec 信号的值
                             ifu_reset_req    ? pc_rtvec :
                          // 否则为顺序取指，则使用当前的 PC 值
                                         pc_r;

  wire ['E203_PC_SIZE-1:0] pc_add_op2 =
                          // 如果是跳转取指，则使用 Simple-BPU 产生的加法操作数二
                             bjp_req ? prdt_pc_add_op2     :
                          // 如果是 reset 后取指，操作数二为 0，则相加后仍等于 pc_rtvec
                             ifu_reset_req    ? 'E203_PC_SIZE'b0 :
                          // 否则为顺序取指，则使用 PC 自增值
```

```
                    pc_incr_ofst ;
 //表示顺序取指令的信号：在没有 reset，没有 flush，不是分支跳转指令的情况下就是顺序取指
  assign ifu_req_seq = (~pipe_flush_req_real) & (~ifu_reset_req) & (~bjp_req);

    // 加法器计算下一条待取指令的 PC 初步值
  assign pc_nxt_pre = pc_add_op1 + pc_add_op2;

  assign pc_nxt =
        //如果 EXU 产生流水线冲刷，则使用 EXU 送过来的新 PC 值（pipe flush pc）
          pipe_flush_req ? {pipe_flush_pc['E203_PC_SIZE-1:1],1'b0} :
          dly_pipe_flush_req ? {pc_r['E203_PC_SIZE-1:1],1'b0} :
        //否则使用前面计算出的 PC 初步值
          {pc_nxt_pre['E203_PC_SIZE-1:1],1'b0};

 ……
   // 产生下一条待取指令的 PC 值
  sirv_gnrl_dfflr #('E203_PC_SIZE) pc_dfflr (pc_ena, pc_nxt, pc_r, clk, rst_n);
```

7.3.5 访问 ITCM 和 BIU

1. 支持 16 位指令

在第 2.2.11 节中介绍了 RISC-V 架构定义的压缩指令子集为 16 位，而蜂鸟 E200 为了提高代码密度选择支持此指令子集，从而会出现程序流中的 32 位和 16 位指令混合交织在一起的情形，而 32 位指令可能处于与 32 位地址边界非对齐的位置。处理此种非对齐的情形成为蜂鸟 E200 IFU 的主要复杂点，其相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
  |----rtl                                      // 存放 RTL 的目录
       |----e203                                // E203 核和 SoC 的 RTL 目录
            |----core                           // 存放 e203 Core 的 RTL 代码
                 |----e203_ifu_ift2icb.v               // 包含非对齐访问逻辑的模块
```

在第 7.1.3 节中介绍了对于非对齐指令取指的常见技术，蜂鸟 E200 采取了其中的剩余缓存（Leftover Buffer）技术。其要点如下。

IFU 每次取指的固定宽度为 32 位，即每次试图取回 32 位的指令字。

- 如果访问的是 ITCM，由于 ITCM 是由 SRAM 构成的，因此上次访问读过 SRAM 之后，SRAM 的输出值会一直保存住（直到下次 SRAM 被再次读或者写过），称之为 Hold-up 住。蜂鸟 E200 的 IFU 会利用 SRAM 输出 Hold-up 的这个特点，而不是将 ITCM 的输出使用 D Flip-Flops 寄存住，此方法可以省略一个 64 比特的寄存器开销。
- 由于 ITCM 的 SRAM 宽度为 64 位，因此其输出为一个与 64 位地址区间对齐的数据，在此称为一个 Lane。假设是地址自增的顺序取指，由于 IFU 每次只取 32 位，因此

会连续两次或者多次在同一个 Lane 里面访问。如果上次已经访问了 ITCM 的 SRAM，下一次取指在同一个 Lane 的访问不会再次真的读 SRAM（即不会打开 SRAM 的 CS 使能），而是利用 SRAM 的 Hold-up 特点，直接使用其保持不变的输出，这样可以省却 SRAM 重复打开造成的动态功耗。

- 如果顺序取指令一个 32 位的指令其非对齐地跨越了 64 位边界，那么会将 SRAM 当前输出的最高 16 位存入 16 比特宽的剩余缓存（Leftover Buffer）之中，并发起新的 ITCM SRAM 访问操作，然后将新访问 ITCM SRAM 返回的低 16 位与剩余缓存中的值拼接成为一个 32 位的完整指令。因此等效于仍然只需要一个周期 ITCM 访问便可取回 32 位指令，不会造成性能损失。
- 如果是非顺序取指（分支跳转或者流水线冲刷等），且地址为非对齐地跨越了 64 位边界，那么就需要连续发起两次 ITCM 读操作。第一次读回数据的高 16 位存入剩余缓存中，第二次读回的低 16 位与剩余缓存中的值拼接成为一个 32 位的完整指令。因此需要两个周期的访问才能取回 32 位指令，会造成额外的一个周期性能损失。由于蜂鸟 E200 的 IFU 并没有设计多体（Bank）化的 ITCM，因此一个周期损失在所难免。由于蜂鸟 E200 是重点关注超低功耗的小面积，因此对此性能点选择放弃。

相关的源代码片段如下所示。

```
// e203_ifu_ift2icb.v 源代码片段

// 处理非对齐取指的主要状态机控制

// State 0: The idle state, means there is no any outstanding ifetch request
localparam ICB_STATE_IDLE = 2'd0;
// State 1: Issued first request and wait response
localparam ICB_STATE_1ST  = 2'd1; //如果非对齐需要发起两次读取操作的第一次读取状态
// State 2: Wait to issue second request
localparam ICB_STATE_WAIT2ND  = 2'd2;//第一次和第二次读取之间的等待状态
// State 3: Issued second request and wait response
localparam ICB_STATE_2ND  = 2'd3; //如果非对齐需要发起两次读取操作的第二次读取状态


wire [ICB_STATE_WIDTH-1:0] icb_state_nxt;
wire [ICB_STATE_WIDTH-1:0] icb_state_r;
wire icb_state_ena;
wire [ICB_STATE_WIDTH-1:0] state_idle_nxt   ;
wire [ICB_STATE_WIDTH-1:0] state_1st_nxt    ;
wire [ICB_STATE_WIDTH-1:0] state_wait2nd_nxt;
wire [ICB_STATE_WIDTH-1:0] state_2nd_nxt    ;
wire state_idle_exit_ena     ;
wire state_1st_exit_ena      ;
wire state_wait2nd_exit_ena  ;
wire state_2nd_exit_ena      ;
```

```
    // Define some common signals and reused later to save gatecounts
    wire icb_sta_is_idle    = (icb_state_r == ICB_STATE_IDLE    );
    wire icb_sta_is_1st     = (icb_state_r == ICB_STATE_1ST     );
    wire icb_sta_is_wait2nd = (icb_state_r == ICB_STATE_WAIT2ND);
    wire icb_sta_is_2nd     = (icb_state_r == ICB_STATE_2ND     );

   ......
  //具体的状态转换请读者自行阅读源代码
  ......

    // The next-state is onehot mux to select different entries
    assign icb_state_nxt =
              ({ICB_STATE_WIDTH{state_idle_exit_ena    }} & state_idle_nxt )
            | ({ICB_STATE_WIDTH{state_1st_exit_ena     }} & state_1st_nxt  )
            | ({ICB_STATE_WIDTH{state_wait2nd_exit_ena}} & state_wait2nd_nxt)
            | ({ICB_STATE_WIDTH{state_2nd_exit_ena     }} & state_2nd_nxt  )
            ;

      sirv_gnrl_dfflr #(ICB_STATE_WIDTH) icb_state_dfflr (icb_state_ena, icb_
state_nxt, icb_state_r, clk, rst_n);

    ......


      // 加载剩余缓存的使能信号
    assign leftover_ena =
           //顺序取指跨界时加载当前 ITCM 输出的最高 16 比特
             holdup2leftover_ena |
           //非顺序取指跨界时发起两次读操作，第一次读操作返回后加载输出的最高 16 比特
             uop1st2leftover_ena;

    assign leftover_nxt =
                       put2leftover_data[15:0] //总是加载输出的最高 16 比特
                       ;
  // 实现剩余缓存的寄存器
    sirv_gnrl_dffl #(16)leftover_dffl(leftover_ena, leftover_nxt,leftover_r,
  clk);
```

上述代码片段只是其代码的很小一部分，e203_ifu_ift2icb 模块可能是蜂鸟 E203 的 IFU 中最复杂的模块，如果感兴趣的读者请自行仔细阅读 GitHub 中的源代码。

2. 生成 ICB 接口访问 ITCM 和 BIU

蜂鸟 E200 的 IFU、ITCM 和 BIU 分开实现，IFU 使用标准的 ICB 协议进行接口。ICB 是蜂鸟 E200 自定义的接口协议，有关此接口协议的详细信息，请参见第 12.2 节。

IFU 生成 ICB 接口访问 ITCM 和 BIU 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
  e200_opensource
      |----rtl                                // 存放 RTL 的目录
          |----e203                           // E203 核和 SoC 的 RTL 目录
```

```
           |----core                  // 存放e203 Core的RTL代码
                  |----e203_ifu_ift2icb.v // 生成ICB接口访问ITCM和BIU的模块
```

其要点如下。

- IFU 有两个 ICB 接口，一个用于访问 ITCM（数据宽度为 64 位），另一个用于访问 BIU（数据宽度为 32 位）。
- 根据 IFU 访问的地址区间进行判断，如果访问的地址落在 ITCM 区间，则通过 ITCM 的 ICB 接口对其进行访问，否则通过 BIU 的 ICB 对外部存储进行访问。

相关源代码片段如下所示。

```
   // e203_ifu_ift2icb.v 源代码片段

 // 访问ITCM的ICB接口
  'ifdef E203_HAS_ITCM //{
  //////////////////////////////////////////////////////////////
  //////////////////////////////////////////////////////////////
  // The ITCM address region indication signal
  input ['E203_ADDR_SIZE-1:0] itcm_region_indic,
  // Bus Interface to ITCM, internal protocol called ICB (Internal Chip Bus)
  //    * Bus cmd channel
  output ifu2itcm_icb_cmd_valid, // Handshake valid
  input  ifu2itcm_icb_cmd_ready, // Handshake ready
         // Note: The data on rdata or wdata channel must be naturally
         //       aligned, this is in line with the AXI definition
  output ['E203_ITCM_ADDR_WIDTH-1:0]  ifu2itcm_icb_cmd_addr,

  //    * Bus RSP channel
  input  ifu2itcm_icb_rsp_valid, // Response valid
  output ifu2itcm_icb_rsp_ready, // Response ready
  input  ifu2itcm_icb_rsp_err,   // Response error
         // Note: the RSP rdata is inline with AXI definition
  input  ['E203_ITCM_DATA_WIDTH-1:0] ifu2itcm_icb_rsp_rdata,

  'endif//}

 // 访问BIU的ICB接口

  'ifdef E203_HAS_MEM_ITF //{
  //////////////////////////////////////////////////////////////
  //////////////////////////////////////////////////////////////
  // Bus Interface to System Memory, internal protocol called ICB (Internal
Chip Bus)
  //    * Bus cmd channel
  output ifu2biu_icb_cmd_valid, // Handshake valid
  input  ifu2biu_icb_cmd_ready, // Handshake ready
         // Note: The data on rdata or wdata channel must be naturally
         //       aligned, this is in line with the AXI definition
  output ['E203_ADDR_SIZE-1:0]  ifu2biu_icb_cmd_addr,
```

```
//     * Bus RSP channel
  input  ifu2biu_icb_rsp_valid, // Response valid
  output ifu2biu_icb_rsp_ready, // Response ready
  input  ifu2biu_icb_rsp_err,   // Response error
           // Note: the RSP rdata is inline with AXI definition
  input  ['E203_SYSMEM_DATA_WIDTH-1:0] ifu2biu_icb_rsp_rdata,

   'endif//}
// 判断地址访问地址区间是否落在 ITCM 区间
  'ifdef E203_HAS_ITCM //{
    //使用比较逻辑比较地址的高位基地址是否与 ITCM 的基地址相等
  assign ifu_icb_cmd2itcm = (ifu_icb_cmd_addr['E203_ITCM_BASE_REGION] == it
cm_region_indic['E203_ITCM_BASE_REGION]);

    //将 ITCM 的 ICB Command Channel 的 valid 信号拉高（如果访问 ITCM）
  assign ifu2itcm_icb_cmd_valid = ifu_icb_cmd_valid & ifu_icb_cmd2itcm;
……
  'endif//}

  'ifdef E203_HAS_MEM_ITF //{
    //如果没有落在 ITCM 则需要访问 BIU
  assign ifu_icb_cmd2biu = 1'b1
          'ifdef E203_HAS_ITCM //{
            & ~(ifu_icb_cmd2itcm)
          'endif//}
            ;

    //将 BIU 的 ICB Command Channel 的 valid 信号拉高（如果访问 BIU）
  wire ifu2biu_icb_cmd_valid_pre  = ifu_icb_cmd_valid & ifu_icb_cmd2biu;
  'endif//}
……
```

7.3.6 ITCM

蜂鸟 E200 采用 ITCM 作为指令存储，如图 7-6 所示，IFU 有专门访问 ITCM 的数据通道（64 位宽），同时 ITCM 也能够被 load、store 指令访问到用于存储数据，因此 ITCM 本身也是 Memory 子系统的重要一部分。有关 ITCM 的微架构细节请参见第 11.4.4 节，本章在此不做赘述。

值得强调的是，蜂鸟 E200 的 ITCM 存储器主体由一块数据宽度为 64 比特的单口 SRAM 组成。ITCM 的大小和基地址（位于全局地址空间中的起始地址）可以通过 config.v 中的宏定义参数配置，请参见第 4.6 节了解相关可配置的信息。

ITCM 采用数据宽度为 64 位能够取得更低的功耗开销，基于如下原因。

- 首先，容量不是特别大的 SRAM，使用宽度为 64 位的 SRAM 在物理大小上比 32 位的 SRAM 面积更加紧凑。因此在同样容量大小下，ITCM 使用 64 位的数据宽度比使用 32 比特的数据宽度面积更小。

- 其次，程序在执行的过程中大多数情形是顺序取指令，而 64 位宽的 ITCM 可以一次取出 64 位的指令流，相比于从 32 位宽的 ITCM 中连续读两次取出 64 位的指令流，只读一次 64 位宽的 SRAM 能够消耗更少的动态功耗。

7.3.7 BIU

如果取指令的地址不落在 ITCM 所在的区间，IFU 则会通过 BIU 访问外部的存储器，有关 BIU 的微架构细节请参见第 12.4 节，在此不做赘述。

7.4 总结

取指是处理器设计非常重要且非常复杂的一部分，为了做到“快”和“连续不断”地取指，尤其是涉及高级的动态分支预测之后，取指部分的设计将会非常复杂和具有挑战性。

处理器微架构设计本身就是一个折中的过程，跟 RISC-V 架构的哲学一样，作者比较欣赏硬件简单就是美的设计哲学。硬件设计应该追求可靠的简单而不是复杂，所谓“好钢用在刀刃上”，只对最常见的情形进行性能优化，而对不常见的情形牺牲其性能而换来硬件结构的简单可靠。

蜂鸟 E200 的取指设计便始终贯穿此设计哲学。蜂鸟 E200 处理器核的设计目标是面向超低功耗的嵌入式处理器核，采取有取有舍的理念。

- 一方面为了低功耗、小面积的目的，舍弃了很多复杂的技术，譬如只采用静态分支预测而未采用动态分支预测，只对 ITCM 访问进行优化，而对 BIU 访问放弃优化。
- 另一方面又保证最常见的情形下性能可观，譬如对 ITCM 区间内的顺序取指，不管地址是否对齐，都能做得到“快”和“连续不断”地取指。

综上原因，蜂鸟 E200 最终的测试基准（Benchmark）跑分在同级别的处理器核中更具竞争力，同时仍然保持了相当小的面积和相当低的功耗。有关蜂鸟 E200 运行测试基准的跑分信息，请参见第 20 章。

第 8 章　一鼓作气，执行力是关键——执行

一鼓作气，再而衰，
三而竭，彼竭我盈也！



在上一章我们介绍了处理器流水线的“取指”单元，在流水线中取指之后便是“译码”和“执行”。执行力是关键，本章将简要介绍处理器的“执行”功能，并介绍蜂鸟 E200 处理器核执行单元（Execution Unit，EXU）的微架构和源码分析。

8.1 执行概述

8.1.1 指令译码

在第 6 章介绍的经典五级流水线中，取指之后的下一级流水线是“译码”（Decode）。由于指令所包含的信息编码在有限长度的指令字中（16 位指令或者 32 位指令），因此需要译码将信息从指令字中翻译出来，常见的信息如下。

- 指令所需要读取的操作数寄存器索引（Index）。
- 指令需要写回的寄存器索引。
- 指令的其他信息，譬如指令类型、指令的操作信息等。

在经典的五级流水线中，在译码阶段直接使用译码出的读操作数寄存器索引，将操作数从通用寄存器组（Regfile）中读取出来。

在此需要顺便提及的是，并非所有的处理器流水线都会在此级读取操作数。如在第 6 章中介绍数据相关性时所介绍的，在目前众多高性能处理器中，普遍采用在每个运算单元前配置乱序发射队列（Issue Queue）的方式，待指令的相关性解除之后从发射队列中发射出来时读取通用寄存器组，然后送给运算单元开始计算。

8.1.2 指令执行

在经典五级流水线中，“译码”且将操作数从通用寄存器组中读取出来后的下一级流水线是“执行”。顾名思义，“执行”便是根据指令的具体操作类型发射给具体的运算单元进行执行，常见的运算单元为以下几种。

- 算术逻辑运算单元（Arithmetic Logical Unit，ALU），主要负责普通逻辑运算、加减法运算和移位运算等基本运算。
- 整数乘法单元，主要负责有符号数或无符号数整数的乘法运算。
- 整数除法单元，主要负责有符号数或无符号数整数的除法运算。
- 浮点运算单元，主要负责浮点指令的运算。由于浮点指令本身种类较多，因此浮点运算单元本身也常分为多个不同的独立运算单元。

以上为常见的运算单元，对于包含特殊指令（或者扩展指令）的处理器核，会相应地增

加特殊的运算单元。

8.1.3 流水线的冲突

除了根据指令的具体类型运行运算之外，指令执行阶段另外一个最重要职能就是维护并解决流水线的冲突，包括资源冲突和数据冲突（包括 WAW、WAR 和 RAW 等数据相关性）。对于流水线冲突的基本概念和常见解决方法在第 6 章已经有所介绍，在此不再赘述。

8.1.4 指令的交付

在经典的五级流水线模型中，处理器的流水线分为取指、译码、执行、访存和写回，其中并没有提及“交付”，但指令的交付（Commit）是处理器微架构中非常重要的一个功能。由于交付的功能阐述清楚需要较多篇幅，因此本书单设一章对其进行详述，请参见第 9 章了解有关交付的详细介绍。

8.1.5 指令发射、派遣、执行、写回的顺序

将指令发射给运算单元，由运算单元进行执行，然后写回的彼此相对顺序，也是执行阶段需要解决的重要问题，此处涉及两个概念。

- 派遣（Dispatch）：可以是按顺序派遣，也可以是乱序派遣。
- 发射（Issue）：可以是按顺序发射，也可以是乱序发射。

对于派遣和发射，由于并没有在经典的五级流水线中提及，有必要先解释其概念。

（1）在处理器设计中，派遣和发射是两个时常被混用的定义。在简单的处理器中，二者往往是说的同一个概念，都是表示指令经过译码之后，被派发到不同的运算单元执行的过程，因此派遣或者发射一般发生在流水线的执行阶段。

- 根据每个周期一次能够发射的指令个数，可以分为“单发射”和“多发射”处理器。“单发射”是指处理器每个周期只能发射一条指令；“多发射”是指处理器每个周期能够发射多条指令，常见的如“双发射”“三发射”或者“四发射”处理器。
- 注意：蜂鸟 E200 处理器核的流水线中使用“派遣”这个名词作为定义。

（2）在一些比较高端的超标量处理器核中，流水线级数甚多，派遣和发射便可能有了不同的含义，派遣往往表示指令经过译码之后被派发到不同的运算单元的等待队列中的过程，而发射往往表示指令从运算单元的等待队列中（解决了数据依赖性之后）发射到运算单元开始执行的过程。

处理器中发射、派遣、执行和写回的顺序是处理器微架构设计中非常重要的一环，根据

顺序的不同，可以分为很多种流派，简述如下。

（1）顺序发射，顺序执行，顺序写回。

- 这种策略往往出现在使用最简单流水线的处理器核中，譬如经典的五级流水线，指令按顺序发射、在运算单元中执行和写回 Regfile。
- 这种策略是性能比较低的做法，硬件实现最简单，面积最小。

（2）顺序发射，乱序执行，顺序写回。

- 由于不同的指令类型往往需要不同的运算单元执行周期，譬如除法指令往往要耗费几十个周期，而最简单的逻辑运算仅需要一个周期便可由 ALU 计算出来，因此如果一味地进行顺序执行，则性能太差。
- 乱序执行便是指在指令的执行阶段可以由不同的运算单元同时执行不同的指令，譬如在除法器执行除法指令期间，ALU 也可以执行其他指令，从而提高性能。
- 但是在最终的写回阶段仍然要严格地按顺序写回，因此很多时候运算单元要等待其他的指令先写回而将其运算单元本身的流水线停滞。

（3）顺序发射，乱序执行，乱序写回。

- 在上述乱序执行的基础上，如果能够让运算单元也乱序地写回，则可以进一步提高性能。
- 运行单元的乱序写回方式可谓门类繁多，可以分为很多种不同的实现，举例如下。

有的处理器会配备重排序缓存（Re-Order Buffer，ROB），因此运算单元一旦执行完毕后，结果就将写回 ROB，而非直接写回 Regfile，最后由 ROB 按顺序写回 Regfile。这是一种典型的乱序写回实现，性能很好，不过这种方案也存在着 ROB 往往因面积过大、数据被腾挪写回两次（先从运算单元到 ROB，再从 ROB 到 Regfile）而增加动态功耗的问题。

有的处理器并不使用 ROB，而是使用统一的物理寄存器组（Physical Register File）实现。由一个统一的物理寄存器组动态地管理逻辑寄存器组的映射关系，运算单元一旦执行完毕后，就将结果乱序地写回物理寄存器组中。此方法相比上述 ROB 方法而言数据只被腾挪一次，因此更加省电，不过控制也相对更加复杂。

有的处理器既没有 ROB，也没有统一的物理寄存器组，但是仍然支持乱序写回。各个运算单元一旦执行完毕后，如果它和其他运算单元中的指令没有数据相关性，便可直接写回 Regfile。

- 乱序写回还可以有很多其他的方法做到，本书限于篇幅在此不加以赘述。

（4）顺序派遣，乱序发射，乱序执行，乱序写回。

- 这种区分了派遣和发射功能的处理器往往属于高性能的超标量处理器。如上文所述，在这种超标量处理器中，指令经过译码后被顺序地派遣到不同运算单元的等待队列中，在等待队列中可以有多条指令等待，待哪一条指令先解决了数据依赖性后便可被先发射到运算单元中开始执行，因此其发射是乱序的。

- 这种高性能处理器往往也会配备 ROB 或者统一的物理寄存器组，因此运算单元的乱序执行和乱序写回可谓小菜一碟。

8.1.6 分支解析

在第 7 章曾经论述了在取指阶段的分支预测功能，对于带条件分支指令，由于其条件的解析需要进行操作数运算（譬如大小比较操作），流水线在取指阶段无法得知该指令的条件跳转结果是跳还是不跳，只能进行预测。

因此在执行阶段，通常需要使用 ALU 对该指令进行条件判断运算（譬如大小比较操作）。ALU 进行条件判断运算的结果将用于解析该分支指令是否真的需要跳转，并且和之前预测的跳转结果进行比对。如果真实的结果和预测的结果不一致，则意味着之前的预测错误，需要进行流水线冲刷（Pipeline Flush），将预测取指（Speculative Fetch）所取的指令都舍弃掉，重新按照真实的跳转方向进行取指。在经典 MIPS 五级流水线中的分支解析过程如图 6-1 所示。

由于分支预测错误造成的流水线冲刷会造成性能损失。流水线级数越深，流水线冲刷造成的性能损失越大。因此理论上来讲，分支解析如果能够发生在比较靠前端（取指）的流水线级数位置，则相对而言其带来的流水线冲刷的性能损失会相对小一些；反之，如果比较靠后，那造成的性能损失就会相对大一些。如何在功能正确且时序能够满足的情况下，尽量在比较靠前端的流水线位置进行分支解析，是处理器微架构设计经常需要考虑的问题。

8.1.7 小结

指令执行阶段的概念相对比较简单、容易理解，但是执行阶段不能够被孤立地看待为简单的一堆运算单元。执行阶段处于衔接前端取指和后端写回的中枢位置，是决定处理器性能高低的主要部分。尤其是在高性能的处理器中，执行阶段是整个动态调度（如第 6 章中介绍过的解决数据相关性的方法）的核心部分，因此应该将执行阶段的功能与整体流水线微架构综合在一起进行理解。

8.2 RISC-V 架构特点对于执行的简化

上一节讨论了处理器执行（包括译码）的相关背景和技术，可以说执行是处理器微架构的中枢核心阶段。在第 2 章中曾经总结性地探讨过 RISC-V 架构追求简化硬件的哲学，具体对于执行（包括译码）而言，RISC-V 架构的如下特点可以大幅简化其硬件实现。

- 规整的指令编码格式。
- 优雅的 16 位指令。
- 精简的指令个数。
- 整数指令都是两操作数。

下文分别予以论述。

8.2.1 规整的指令编码格式

得益于后发优势和总结了多年来处理器发展的教训，RISC-V 的指令集编码非常规整，指令所需的通用寄存器的索引（Index）都被放在固定的位置。因此指令译码器（Instruction Decoder）可以非常便捷地译码出寄存器索引，然后从通用寄存器组中读取出操作数，同样也可以很容易地译码出指令的类型和具体信息。

8.2.2 优雅的 16 位指令

RISC-V 架构为了提高代码密度，而定义了一种可选的压缩（Compressed）指令子集，由字母 C 表示。RISC-V 架构的一个精妙之处在于每一条 16 位长的指令都能找到一一对应的原始 32 位指令。因此译码逻辑可以利用此特点将 16 位指令展开成为其对应的 32 位指令，从而使得流水线后续部分看到的都是统一的 32 位指令，执行阶段无须区分指令是 16 位指令还是 32 位指令。

8.2.3 精简的指令个数

RISC-V 架构的指令集数目非常简洁，基本的 RISC-V 指令数目仅有 40 多条，加上其他的模块化扩展指令总共几十条指令。指令数目精简也就意味着只需处理更少的情形，简化执行阶段的硬件设计负担。

关于蜂鸟 E200 系列支持的 RISC-V 指令列表，详见附录 A。

8.2.4 整数指令都是两操作数

RISC-V 的整数指令的操作数个数均是规整的 1 操作数或者 2 操作数指令，没有 3 操作数指令，这样可以简化操作数读取和数据相关性检测部分的硬件设计。

8.3 蜂鸟 E200 处理器的执行实现

如第 6 章所介绍，蜂鸟 E200 是两级流水线架构，其“译码”“执行”“交付”和“写回”功能均处于流水线的第二级，由 EXU 单元完成，如图 8-1 所示。

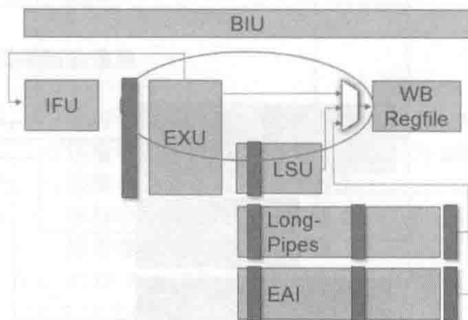


图 8-1　蜂鸟 E200 流水线中的 EXU 单元

8.3.1 执行指令列表

蜂鸟 E200 系列支持的所有 RISC-V 指令集均需由 EXU 进行译码、派遣和写回。关于蜂鸟 E200 系列支持的 RISC-V 指令列表，详见附录 A。

8.3.2 EXU 总体设计思路

蜂鸟 E200 的 EXU 单元微架构如图 8-2 所示，其主要功能包括以下情况。

- 将 IFU 通过 IR 寄存器发送给 EXU 的指令进行译码和派遣（如图 8-2 中的“译码与派遣”）。
- 通过译码出的操作数寄存器索引（Index）读取 Regfile（如图 8-2 中的 RD-Regfile）。
- 维护指令的数据相关性（如图 8-2 中的 OITF）。
- 将指令派遣（Dispatch）给不同的运算单元执行（如图 8-2 中的 ALU、Long-Pipes、LSU 以及 EAI）。
- 将指令交付（如图 8-2 中的交付）。
- 将指令运算的结果写回 Regfile（如图 8-2 中的 WB Arb）。

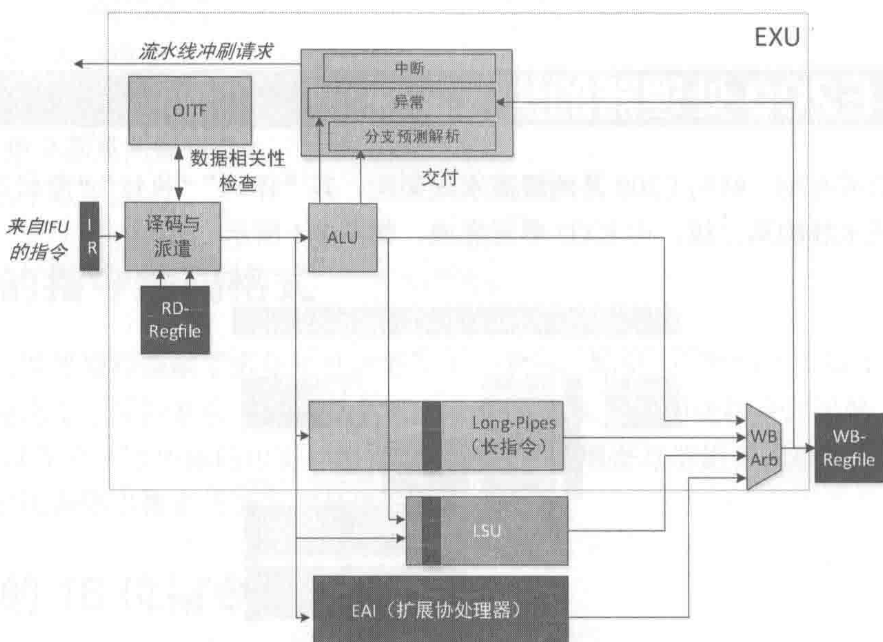


图 8-2　蜂鸟 E200 处理器核 EXU 微架构示意图

下文对 EXU 的不同子模块分别予以论述。

8.3.3　译码

译码（Decode）模块主要用于对 IR 寄存器中的指令进行译码，要点如下。

- 译码的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                // 存放 RTL 的目录
        |----e203                           // E203 核和 SoC 的 RTL 目录
            |----core                       // 存放 e203 Core 的 RTL 代码
                |----e203_exu_decode.v                  // Decode 模块
```

- 译码模块完全由组合逻辑组成。其主要逻辑即根据 RISC-V 指令的编码规则进行译码，产生不同的指令类型信息，操作数寄存器索引等。相关源代码片段如下所示。

```
// e203_exu_decode.v 源代码片段

module e203_exu_decode(

  //////////////////////////////////////////////////////////////
  // The IR stage to Decoder
```

```
//以下为来自 IFU 输入 EXU Decode 模块的信号

  input  ['E203_INSTR_SIZE-1:0] i_instr, // 来自于 IFU 的 32 位指令
  input  ['E203_PC_SIZE-1:0] i_pc,       // 来自于 IFU 的当前指令对应 PC 值
  input  i_prdt_taken,
  input  i_misalgn,                 // 表明当前指令遭遇了取指非对齐异常
  input  i_buserr,                  //表明当前指令遭遇了取指存储器访问错误

  //////////////////////////////////////////////////////////////////
  // The Decoded Info-Bus

//以下为对指令进行译码得到的信息

  output dec_rs1x0, //该指令源操作数 1 的寄存器索引为 x0
  output dec_rs2x0, //该指令源操作数 2 的寄存器索引为 x0
  output dec_rs1en, //该指令需要读取源操作数 1
  output dec_rs2en, //该指令需要读取源操作数 2
  output dec_rdwen, //该指令需要写结果操作数
  output ['E203_RFIDX_WIDTH-1:0] dec_rs1idx,//该指令源操作数 1 的寄存器索引
  output ['E203_RFIDX_WIDTH-1:0] dec_rs2idx,//该指令源操作数 2 的寄存器索引
  output ['E203_RFIDX_WIDTH-1:0] dec_rdidx, //该指令结果寄存器索引
  output ['E203_DECINFO_WIDTH-1:0] dec_info,//该指令的其他信息，将其打包成为一组
                                          //宽信号，称之为信息总线（info bus）
  output ['E203_XLEN-1:0] dec_imm,          //该指令使用的立即数的值
......
  output dec_ilegl, //经过译码后，发现本指令是个非法指令
......


//以下为译码器的部分关键代码解析以方便读者理解。
对于 32 位指令的译码比较直接，因为指令编码比较规整。而对于 16 位指令的译码相对比较复杂，因为
指令编码没有 32 位指令规整

//该指令为 32 位指令还是 16 位指令的指示信号
wire rv32 = (~(i_instr[4:2] == 3'b111)) & opcode_1_0_11;
......
//取出 32 位指令的关键编码段
  wire [4:0]  rv32_rd      = rv32_instr[11:7];//32 位指令的结果操作数索引
  wire [2:0]  rv32_func3   = rv32_instr[14:12];//32 位指令的 func3 段
  wire [4:0]  rv32_rs1     = rv32_instr[19:15]; //32 位指令的源操作数 1 索引
  wire [4:0]  rv32_rs2     = rv32_instr[24:20]; //32 位指令的源操作数 2 索引
  wire [6:0]  rv32_func7   = rv32_instr[31:25]; //32 位指令的 func7 段

//同理，取出 16 位指令的关键编码段
  wire [4:0]  rv16_rd      = rv32_rd;
  wire [4:0]  rv16_rs1     = rv16_rd;
  wire [4:0]  rv16_rs2     = rv32_instr[6:2];

  wire [4:0]  rv16_rdd     = {2'b01,rv32_instr[4:2]};
```

```
  wire [4:0]  rv16_rss1   = {2'b01,rv32_instr[9:7]};
  wire [4:0]  rv16_rss2   = rv16_rdd;

  wire [2:0] rv16_func3   = rv32_instr[15:13];

//以下为对 32 位指令的指令类型译码
  wire rv32_load      = opcode_6_5_00 & opcode_4_2_000 & opcode_1_0_11;
  wire rv32_store     = opcode_6_5_01 & opcode_4_2_000 & opcode_1_0_11;
  wire rv32_madd      = opcode_6_5_10 & opcode_4_2_000 & opcode_1_0_11;
  wire rv32_branch    = opcode_6_5_11 & opcode_4_2_000 & opcode_1_0_11;

  wire rv32_load_fp   = opcode_6_5_00 & opcode_4_2_001 & opcode_1_0_11;
  wire rv32_store_fp  = opcode_6_5_01 & opcode_4_2_001 & opcode_1_0_11;
  ......

//同理，以下为对 16 位指令的指令类型译码
  wire rv16_addi4spn      = opcode_1_0_00 & rv16_func3_000;//
  wire rv16_lw            = opcode_1_0_00 & rv16_func3_010;//
  wire rv16_sw            = opcode_1_0_00 & rv16_func3_110;//

  wire rv16_addi          = opcode_1_0_01 & rv16_func3_000;//
  wire rv16_jal           = opcode_1_0_01 & rv16_func3_001;//
  wire rv16_li            = opcode_1_0_01 & rv16_func3_010;//
......
  wire rv16_swsp          = opcode_1_0_10 & rv16_func3_110;//

//生成 BJP 单元所需的信息总线（Info Bus）。BJP 单元是 ALU 的一个子单元，参见第 8.3.8 节对
ALU 的详细介绍。
  wire bjp_op = dec_bjp | rv32_mret | (rv32_dret & (~rv32_dret_ilgl)) | rv3
2_fence_fencei;

  wire ['E203_DECINFO_BJP_WIDTH-1:0] bjp_info_bus;
  assign bjp_info_bus['E203_DECINFO_GRP      ]  = 'E203_DECINFO_GRP_BJP;
  assign bjp_info_bus['E203_DECINFO_RV32     ]  = rv32;
  assign bjp_info_bus['E203_DECINFO_BJP_JUMP ]  = dec_jal | dec_jalr;
  assign bjp_info_bus['E203_DECINFO_BJP_BPRDT]  = i_prdt_taken;
  assign bjp_info_bus['E203_DECINFO_BJP_BEQ  ]  = rv32_beq | rv16_beqz;
  assign bjp_info_bus['E203_DECINFO_BJP_BNE  ]  = rv32_bne | rv16_bnez;
......

//生成 Regular ALU 单元所需的信息总线（Info Bus）。Regular ALU 单元为 ALU 的一个子单元，
参见第 8.3.8 节对 ALU 的详细介绍。
  wire alu_op = (~rv32_sxxi_shamt_ilgl) & (~rv16_sxxi_shamt_ilgl)
......
  wire need_imm;
  wire ['E203_DECINFO_ALU_WIDTH-1:0] alu_info_bus;
  assign alu_info_bus['E203_DECINFO_GRP    ]    = 'E203_DECINFO_GRP_ALU;
  assign alu_info_bus['E203_DECINFO_RV32   ]    = rv32;
......
  assign alu_info_bus['E203_DECINFO_ALU_SUB]    = rv32_sub  | rv16_sub;
  assign alu_info_bus['E203_DECINFO_ALU_SLT]    = rv32_slt  | rv32_slti;
  assign alu_info_bus['E203_DECINFO_ALU_SLTU]   = rv32_sltu | rv32_sltiu;
```

```
......

//生成 CSR 单元所需的信息总线（Info Bus）。CSR 单元为 ALU 的一个子单元，参见第 8.3.8 节对
ALU 的详细介绍。
    wire csr_op = rv32_csr;
    wire ['E203_DECINFO_CSR_WIDTH-1:0] csr_info_bus;
    assign csr_info_bus['E203_DECINFO_GRP    ]    = 'E203_DECINFO_GRP_CSR;
    assign csr_info_bus['E203_DECINFO_RV32   ]    = rv32;
    assign csr_info_bus['E203_DECINFO_CSR_CSRRW ] = rv32_csrrw | rv32_csrrwi;
    assign csr_info_bus['E203_DECINFO_CSR_CSRRS ] = rv32_csrrs | rv32_csrrsi;
    assign csr_info_bus['E203_DECINFO_CSR_CSRRC ] = rv32_csrrc | rv32_csrrci;
    assign csr_info_bus['E203_DECINFO_CSR_RS1IMM] = rv32_csrrwi | rv32_csrrsi
 | rv32_csrrci;
    assign csr_info_bus['E203_DECINFO_CSR_ZIMMM ] = rv32_rs1;
    assign csr_info_bus['E203_DECINFO_CSR_RS1IS0] = rv32_rs1_x0;
    assign csr_info_bus['E203_DECINFO_CSR_CSRIDX] = rv32_instr[31:20];
  ......

  //生成乘除法单元所需的信息总线（Info Bus），参见第 8.3.8 节对 ALU 的详细介绍。
    wire muldiv_op = rv32_op & rv32_func7_0000001;

    wire ['E203_DECINFO_MULDIV_WIDTH-1:0] muldiv_info_bus;
    assign muldiv_info_bus['E203_DECINFO_GRP          ] = 'E203_DECINFO_GRP_MULDIV;
    assign muldiv_info_bus['E203_DECINFO_RV32         ] = rv32       ;
    assign muldiv_info_bus['E203_DECINFO_MULDIV_MUL   ] = rv32_mul   ;
    assign muldiv_info_bus['E203_DECINFO_MULDIV_MULH  ] = rv32_mulh  ;
    assign muldiv_info_bus['E203_DECINFO_MULDIV_MULHSU] = rv32_mulhsu ;
    assign muldiv_info_bus['E203_DECINFO_MULDIV_MULHU ] = rv32_mulhu  ;
  ......

  //生成 AGU 单元所需的信息总线(Info Bus)。AGU 单元是 ALU 的一个子单元，用于处理 AMO 和 Load、
Store 指令，参见第 8.3.8 节对 ALU 的详细介绍。
    wire   amoldst_op = rv32_amo | rv32_load | rv32_store | rv16_lw | rv16_sw
 | (rv16_lwsp & (~rv16_lwsp_ilgl)) | rv16_swsp;

    wire ['E203_DECINFO_AGU_WIDTH-1:0] agu_info_bus;
    assign agu_info_bus['E203_DECINFO_GRP    ] = 'E203_DECINFO_GRP_AGU;
    assign agu_info_bus['E203_DECINFO_RV32   ] = rv32;
    assign agu_info_bus['E203_DECINFO_AGU_LOAD   ] = rv32_load   | rv32_lr_w |
 rv16_lw | rv16_lwsp;
    assign agu_info_bus['E203_DECINFO_AGU_STORE  ] = rv32_store | rv32_sc_w |
 rv16_sw | rv16_swsp;
    assign agu_info_bus['E203_DECINFO_AGU_SIZE   ] = lsu_info_size;
    assign agu_info_bus['E203_DECINFO_AGU_USIGN  ] = lsu_info_usign;
    assign agu_info_bus['E203_DECINFO_AGU_EXCL   ] = rv32_lr_w | rv32_sc_w;
    assign agu_info_bus['E203_DECINFO_AGU_AMO    ] = rv32_amo & (~(rv32_lr_w
 | rv32_sc_w));// We seperated the EXCL out of AMO in LSU handling
    assign agu_info_bus['E203_DECINFO_AGU_AMOSWAP] = rv32_amoswap_w;
    assign agu_info_bus['E203_DECINFO_AGU_AMOADD ] = rv32_amoadd_w ;
    assign agu_info_bus['E203_DECINFO_AGU_AMOAND ] = rv32_amoand_w ;
  ......
```

```
    //以下逻辑是典型的 5 输入并行多路选择器（使用 Verilog assign 语法编码 And-Or 逻辑），选择信号是指令类型信号（譬如 alu_op，bjp_op 等），从而根据不同的指令分组，将它们的信息总线经过并行多路选择的方式复用到一个统一的输出信号 dec_info 上。
      assign dec_info =
                 ({'E203_DECINFO_WIDTH{alu_op}}     & {{'E203_DECINFO_WIDTH-'E203_DECINFO_ALU_WIDTH{1'b0}},alu_info_bus})
               | ({'E203_DECINFO_WIDTH{amoldst_op}} & {{'E203_DECINFO_WIDTH-'E203_DECINFO_AGU_WIDTH{1'b0}},agu_info_bus})
               | ({'E203_DECINFO_WIDTH{bjp_op}}     & {{'E203_DECINFO_WIDTH-'E203_DECINFO_BJP_WIDTH{1'b0}},bjp_info_bus})
               | ({'E203_DECINFO_WIDTH{csr_op}}     & {{'E203_DECINFO_WIDTH-'E203_DECINFO_CSR_WIDTH{1'b0}},csr_info_bus})
               | ({'E203_DECINFO_WIDTH{muldiv_op}}  & {{'E203_DECINFO_WIDTH-'E203_DECINFO_CSR_WIDTH{1'b0}},muldiv_info_bus})
                 ;


  //以下译码是否需要读取寄存器操作数 1，寄存器操作数 2，是否需要写结果寄存器

   // All the RV32IMA need RD register except the
   //   * Branch, Store,
   //   * fence, fence_i
   //   * ecall, ebreak
   wire rv32_need_rd =
                     (~rv32_rd_x0) & (
                    (
                     (~rv32_branch) & (~rv32_store)
                   & (~rv32_fence_fencei)
                   & (~rv32_ecall_ebreak_ret_wfi)
                    )
                  );

......
   wire rv32_need_rs1 =
                  ......

   wire rv32_need_rs2 = (~rv32_rs2_x0) & (
                 ......

     //以下译码出指令的立即数，不同的指令类型有不同的立即数编码形式需要译码。

  //首先译码 32 位指令的不同立即数格式
   wire [31:0]  rv32_i_imm = {
                             {20{rv32_instr[31]}}
                           , rv32_instr[31:20]
                           };
                        ......
   wire [31:0]  rv32_s_imm = {
                          ......

   wire [31:0]  rv32_b_imm = {
```

```
                                ……
    wire [31:0]  rv32_u_imm = {
                                ……
    wire [31:0]  rv32_j_imm = {
                                ……
```

//其次译码 16 位指令的不同立即数格式

```
    wire rv16_imm_sel_cis = rv16_lwsp;
    wire [31:0]  rv16_cis_imm ={
                       24'b0
                     , rv16_instr[3:2]
                     , rv16_instr[12]
                     , rv16_instr[6:4]
                     , 2'b0
                      };
                                ……
    wire [31:0]  rv16_cis_d_imm ={
                                ……
    wire [31:0]  rv16_cili_imm ={
                                ……
    wire [31:0]  rv16_cilui_imm ={
                                ……
    wire [31:0]  rv16_ci16sp_imm ={
                                ……
```

//以下逻辑是典型的 5 输入并行多路选择器（使用 Verilog assign 语法编码 And-Or 逻辑），根据不同的 32 位立即数类型，选择生成 32 位指令最终的立即数。

```
    wire [31:0]  rv32_imm =
                   ({32{rv32_imm_sel_i}} & rv32_i_imm)
                 | ({32{rv32_imm_sel_s}} & rv32_s_imm)
                 | ({32{rv32_imm_sel_b}} & rv32_b_imm)
                 | ({32{rv32_imm_sel_u}} & rv32_u_imm)
                 | ({32{rv32_imm_sel_j}} & rv32_j_imm)
                 ;
```

//以下逻辑是典型的 10 输入并行多路选择器（使用 Verilog assign 语法编码 And-Or 逻辑），根据不同的 16 位立即数类型，选择生成 16 位指令最终的立即数。

```
    wire [31:0]  rv16_imm =
                   ({32{rv16_imm_sel_cis   }} & rv16_cis_imm)
                 | ({32{rv16_imm_sel_cili  }} & rv16_cili_imm)
                 | ({32{rv16_imm_sel_cilui }} & rv16_cilui_imm)
                 | ({32{rv16_imm_sel_ci16sp}} & rv16_ci16sp_imm)
                 | ({32{rv16_imm_sel_css   }} & rv16_css_imm)
                 | ({32{rv16_imm_sel_ciw   }} & rv16_ciw_imm)
                 | ({32{rv16_imm_sel_cl    }} & rv16_cl_imm)
```

```
                              | ({32{rv16_imm_sel_cs    }} & rv16_cs_imm)
                              | ({32{rv16_imm_sel_cb    }} & rv16_cb_imm)
                              | ({32{rv16_imm_sel_cj    }} & rv16_cj_imm)
                              ;

  //根据指令是 16 位还是 32 位宽度，选择生成最终的立即数
   assign dec_imm = rv32 ? rv32_imm : rv16_imm;

  //根据指令是 16 位还是 32 位宽度，选择生成最终的操作数寄存器索引
   assign dec_rs1idx = rv32 ? rv32_rs1['E203_RFIDX_WIDTH-1:0] : rv16_rs1idx;
   assign dec_rs2idx = rv32 ? rv32_rs2['E203_RFIDX_WIDTH-1:0] : rv16_rs2idx;
   assign dec_rdidx  = rv32 ? rv32_rd ['E203_RFIDX_WIDTH-1:0] : rv16_rdidx ;
  ......

  //译码出不同的非法指令情形
  assign dec_ilegl =
             (rv_all0s1s_ilgl)
           | (rv_index_ilgl)
           | (rv16_addi16sp_ilgl)
           | (rv16_addi4spn_ilgl)
           | (rv16_li_lui_ilgl)
           | (rv16_sxxi_shamt_ilgl)
           | (rv32_sxxi_shamt_ilgl)
           | (rv32_dret_ilgl)
           | (rv16_lwsp_ilgl)
           | (~legl_ops);
```

8.3.4 整数通用寄存器组

整数通用寄存器组（Integer Register File，简称 Integer-Regfile）模块主要用于实现 RISC-V 架构定义的整数通用寄存器组。请参见附录 A4.1 节了解有关 RISC-V 架构通用寄存器组的信息。

RISC-V 的整数指令都是单操作数或者两操作数指令，且蜂鸟 E200 属于单发射（一次发射派遣一条指令）的微架构，因此 Integer-Regfile 模块只需要支持最多两个读端口。同时，蜂鸟 E200 的写回策略是按顺序每次写回一条指令的微架构，因此 Integer-Regfile 模块只需要支持一个写端口。

基于以上要点，蜂鸟 E200 的 Integer-Regfile 模块微架构如图 8-3 所示，要点如下。

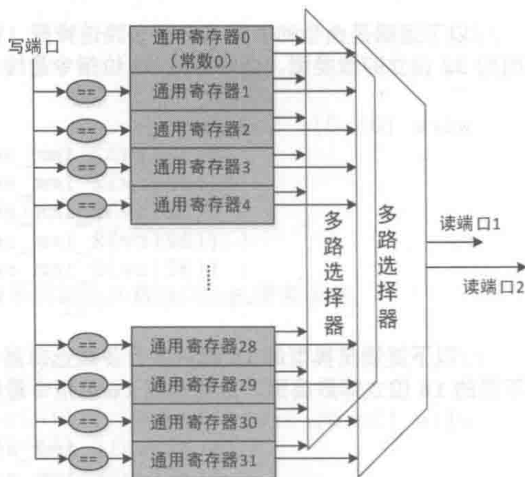


图 8-3 蜂鸟 E200 处理器核 Integer-Regfile 微架构图

- Integer-Regfile 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                    // 存放RTL的目录
        |----e203                               // E203核和SoC的RTL目录
            |----core                           // 存放e203 Core的RTL代码
                |----e203_exu_regfile.v             // Integer Regfile模块
```

- 如图 8-3 所示，Integer-Regfile 的写端口逻辑将输入的结果寄存器索引和各自的寄存器号进行比较，产生写使能信号，被使能的通用寄存器即将写数据写入寄存器（x0 由于是常数，因此无须写入）。
- Integer-Regfile 的每个读端口都是一个纯粹的并行多路选择器，多路选择器的选择信号即读操作数的寄存器索引。为了减少功耗，读端口的寄存器索引信号（用于并行多路选择器的选择信号）被专用的寄存器进行寄存，只有在执行需要读操作数的指令时才会被加载（否则保持不变），从而减少读端口的动态翻转功耗。请参见第 15 章了解更多低功耗设计的诀窍。
- Integer-Regfile 有两个可配置选项，可以通过 config.v 中的宏进行配置。
  通用寄存器的个数可以由宏 E203_CFG_REGNUM_IS_32 或者 E203_CFG_REGNUM_IS_16 进行制定为 32 个（即为 RV32I 架构）或者 16 个（即为 RV32E 架构）。
  通用寄存器可以用锁存器（Latch）实现，可以由宏 E203_CFG_REGFILE_LATCH_BASED 进行配置。如果没有定义此宏，通用寄存器则由 D Flip-Flops（DFF）实现。使用锁存器能够显著地减少 Integer-Regfile 的面积和功耗，但是也会给 ASIC 流程带来某些困难，用户可以自行选择是否使用。
- config.v 文件在 e200_opensource 目录的结构如下。请参见第 4.6 节了解更多蜂鸟 E200 的可配置信息。

```
e200_opensource
    |----rtl                                    // 存放RTL的目录
        |----e203                               // E203核和SoC的RTL目录
            |----core                           // 存放core相关模块的RTL代码
                |----config.v                   // 设定配置的源文件
```

- Integer-Regfile 模块的相关源代码片段如下所示。

```
// e203_exu_regfile.v 源代码片段


//使用二维数组定义通用寄存器组
  wire ['E203_XLEN-1:0] rf_r ['E203_RFREG_NUM-1:0];
  wire ['E203_RFREG_NUM-1:0] rf_wen;

  'ifdef E203_REGFILE_LATCH_BASED //{
//如果使用锁存器实现通用寄存器。则需要将写端口使用DFF专门寄存一拍，此举能够防止锁存器带来的写端口至读端口之间的锁存器穿通效应。
  // Use DFF to buffer the write-port
```

```
    wire ['E203_XLEN-1:0] wbck_dest_dat_r;
    sirv_gnrl_dffl #('E203_XLEN)       wbck_dat_dffl (wbck_dest_wen, wbck_de
st_dat, wbck_dest_dat_r, clk);
    wire ['E203_RFREG_NUM-1:0] clk_rf_ltch;
    'endif//}

    genvar i;
    generate //{//通过使用参数化的 generate 语法生成 Regfile 的逻辑

      for (i=0; i<'E203_RFREG_NUM; i=i+1) begin:regfile//{

        if(i==0) begin: rf0
          //x0 是一个常数 0，因此无须产生写逻辑。
          // x0 cannot be wrote since it is constant-zeros
          assign rf_wen[i] = 1'b0;
          assign rf_r[i]= 'E203_XLEN'b0;
         'ifdef E203_REGFILE_LATCH_BASED //{
          assign clk_rf_ltch[i] = 1'b0;
         'endif//}
        end
        else begin: rfno0
          //通过对写结果寄存器的索引号和寄存器号进行比较产生写使能。
          assign rf_wen[i] = wbck_dest_wen & (wbck_dest_idx == i) ;
         'ifdef E203_REGFILE_LATCH_BASED //{
            //如果是使用锁存器的配置，则人为的明确为每一个通用寄存器配置一个门控时钟以
            //节省功耗
          e203_clkgate u_e203_clkgate(
            .clk_in  (clk  ),
            .test_mode(test_mode),
            .clock_en(rf_wen[i]),
            .clk_out (clk_rf_ltch[i])
          );
             //如果是使用锁存器的配置，则例化锁存器实现通用寄存器
             //from write-enable to clk_rf_ltch to rf_ltch
          sirv_gnrl_ltch #('E203_XLEN) rf_ltch (clk_rf_ltch[i], wbck_dest
_dat_r, rf_r[i]);
         'else//}{
             //如果是不使用锁存器的配置，则例化 DFF 实现通用寄存器
                         // 由于此处有明确的 Load-enable 信号，综合工具会自动插
入门控时钟（ICG）以节省功耗。有关综合工具自动插入门控时钟的更多信息，请参见第 15.1.5 节。

          sirv_gnrl_dffl #('E203_XLEN) rf_dffl (rf_wen[i], wbck_dest_dat,
 rf_r[i], clk);
         'endif//}
        end

      end//}
    endgenerate//}

      //每个读端口都是一个纯粹的并行多路选择器
      //多路选择器的选择信号即读操作数的寄存器索引
```

```
assign read_src1_dat = rf_r[read_src1_idx];
assign read_src2_dat = rf_r[read_src2_idx];
```

8.3.5 CSR 寄存器

在第 2 章中曾经介绍过，RISC-V 的架构中定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用其自己的地址编码空间，与存储器寻址的地址区间完全无关系。请参见附录 B 了解 CSR 寄存器的列表与详细信息。

CSR 寄存器的访问采用专用的 CSR 读写指令，包括 CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI、CSRRCI 指令，请参见附录 A14.2 节了解指令的具体信息。

蜂鸟 E200 的 EXU 单元中的 CSR 寄存器模块主要用于实现蜂鸟 E200 所支持的 CSR 寄存器功能，其要点如下。

- CSR 寄存器模块的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
     |----rtl                                // 存放 RTL 的目录
          |----e203                          // E203 核和 SoC 的 RTL 目录
               |----core                     // 存放 e203 Core 的 RTL 代码
                    |----e203_exu_csr.v            // CSR 寄存器模块
```

- 在 ALU 模块中的 CSR 读写控制模块会产生 CSR 读写控制信号（参见第 8.3.8 节对 ALU 模块中 CSR 读写控制模块的介绍），而 CSR 寄存器模块则严格按照 RISC-V 架构的定义实现各个 CSR 寄存器的具体功能。其相关源代码片段如下所示。

```
// e203_exu_csr.v 源代码片段

  input csr_ena,     //来自于 ALU 的 CSR 读写使能信号
  input csr_wr_en, //CSR 写操作指示信号
  input csr_rd_en,     //CSR 读操作指示信号
  input [12-1:0] csr_idx,//CSR 寄存器的地址索引

  output ['E203_XLEN-1:0] read_csr_dat,//读操作从 CSR 寄存器模块中读出的数据
  input  ['E203_XLEN-1:0] wbck_csr_dat,//写操作写入 CSR 寄存器模块的数据

  ......
//以 MTVEC 寄存器为例

//实现 MTVEC 寄存器
//0x305 MRW mtvec Machine trap-handler base address.
wire sel_mtvec = (csr_idx == 12'h305); //对 CSR 寄存器索引进行译码判断是否选中 mtvec
wire rd_mtvec = csr_rd_en & sel_mtvec;
```

```
    wire wr_mtvec = sel_mtvec & csr_wr_en;
    wire mtvec_ena = (wr_mtvec & wbck_csr_wen); //产生写mtvec使能信号
    wire ['E203_XLEN-1:0] mtvec_r;
    wire ['E203_XLEN-1:0] mtvec_nxt = wbck_csr_dat;
            //例化生成mtvec寄存器的DFF
    sirv_gnrl_dfflr #('E203_XLEN) mtvec_dfflr (mtvec_ena, mtvec_nxt, mtvec_r,
clk, rst_n);
    wire ['E203_XLEN-1:0] csr_mtvec = mtvec_r;
    ......

    // 注意：
    //  对于读地址不存在的CSR寄存器，返回数据为0。写地址不存在的CSR寄存器，则忽略此写操作。
    //  蜂鸟E200对于CSR访问不会产生异常。


    //生成CSR读操作所需的读数据，本质上该逻辑是使用and-or方式实现的并行多路选择器。
    assign read_csr_dat = 'E203_XLEN'b0
                 | ({'E203_XLEN{rd_mstatus  }} & csr_mstatus  )
                 | ({'E203_XLEN{rd_mie      }} & csr_mie      )
                 | ({'E203_XLEN{rd_mtvec    }} & csr_mtvec    )
                 | ({'E203_XLEN{rd_mepc     }} & csr_mepc     )
                 | ({'E203_XLEN{rd_mscratch }} & csr_mscratch )
                 | ({'E203_XLEN{rd_mcause   }} & csr_mcause   )
                 | ({'E203_XLEN{rd_mbadaddr }} & csr_mbadaddr )
                 | ({'E203_XLEN{rd_mip      }} & csr_mip      )
                 | ({'E203_XLEN{rd_misa     }} & csr_misa      )
                 | ({'E203_XLEN{rd_mvendorid}} & csr_mvendorid)
                 | ({'E203_XLEN{rd_marchid  }} & csr_marchid  )
                 | ({'E203_XLEN{rd_mimpid   }} & csr_mimpid   )
                 | ({'E203_XLEN{rd_mhartid  }} & csr_mhartid  )
                 | ({'E203_XLEN{rd_mcycle   }} & csr_mcycle   )
                 | ({'E203_XLEN{rd_mcycleh  }} & csr_mcycleh  )
                 | ({'E203_XLEN{rd_minstret }} & csr_minstret )
                 | ({'E203_XLEN{rd_minstreth}} & csr_minstreth)
                 | ({'E203_XLEN{rd_mcounterstop}} & csr_mcounterstop)
                 | ({'E203_XLEN{rd_mcgstop}} & csr_mcgstop)
                 | ({'E203_XLEN{rd_dcsr      }} & csr_dcsr     )
                 | ({'E203_XLEN{rd_dpc       }} & csr_dpc      )
                 | ({'E203_XLEN{rd_dscratch }} & csr_dscratch)
                 ;
    ......
```

- 蜂鸟E200在标准的RISC-V架构定义的基础上，还添加了若干自定义的CSR寄存器，关于蜂鸟E200自定义的CSR寄存器列表，详见附录B3。

8.3.6 指令发射派遣

在第8.1.5节已经详细介绍了“发射”和“派遣”的概念，在此不再赘述。

由于蜂鸟E200是简单的两级流水线微架构，派遣（Dispatch）或者发射（Issue）发生在流水线的执行阶段，指的是同一个概念，即表示指令经过译码且从寄存器组中读取了操作

数之后被派发到不同的运算单元执行的过程。蜂鸟 E200 处理器核的流水线中使用“派遣”这个名词作为定义，因此本节之后将统一使用“派遣”进行微架构详细介绍。

蜂鸟 E200 的派遣功能由派遣模块和 ALU 模块联合完成，要点如下。

- 派遣和 ALU 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                    // 存放 RTL 的目录
         |----e203                              // E203 核和 SoC 的 RTL 目录
              |----core                         // 存放 e203 Core 的 RTL 代码
                   |----e203_exu_disp.v             // Dispatch 模块
                   |----e203_exu_alu.v              // ALU 模块
```

- 蜂鸟 E200 执行阶段的派遣机制特别之处如下。

  其所有指令都必须被派遣给 ALU，并且通过 ALU 与交付模块的接口进行交付，有关交付的详细信息请参见第 9 章。

  如果是长指令，也需通过 ALU 进一步将其发送至相应的长指令运算单元。譬如，属于长指令类型的 Load/Store 指令便是通过 ALU 的 AGU 子单元被进一步发送至 LSU 单元进行执行。有关长指令的定义请参见第 8.3.7 节，有关 AGU 和 LSU 的实现请参见第 11.4.2 和 11.4.3 节。

- 由于所有的指令都需要通过 ALU，因此实际的派遣功能发生在 ALU 的内部。由于蜂鸟 E200 的译码模块在进行译码时，已经根据执行指令的运算单元进行了分组，并且译码出了其相应的指示信号，所以可以按照其指示信号将指令派遣给相应的运算单元。相关源代码片段如下。

```
// e203_exu_disp.v 源代码片段


//将指令派遣给 ALU 的接口采用 valid-ready 模式的握手信号。由于所有的指令都会被派遣给 ALU，
//因此此处直接对接。
  wire   disp_i_ready_pos = disp_o_alu_ready;
  assign disp_o_alu_valid = disp_i_valid_pos;

//将操作数派遣给 ALU
  assign disp_o_alu_rs1   = disp_i_rs1_msked;
  assign disp_o_alu_rs2   = disp_i_rs2_msked;

//将指令信息派遣给 ALU
  assign disp_o_alu_rdwen = disp_i_rdwen;//指令是否写回结果寄存器
  assign disp_o_alu_rdidx = disp_i_rdidx; //指令写回的结果寄存器索引
  assign disp_o_alu_info  = disp_i_info;  //指令的信息（Info Bus）


  assign disp_o_alu_imm  = disp_i_imm;//指令使用的立即数的值
```

```
    assign disp_o_alu_pc      = disp_i_pc;//该指令的 PC

    assign disp_o_alu_misalgn= disp_i_misalgn;//该指令取指时发生了非对齐错误
    assign disp_o_alu_buserr = disp_i_buserr ; //该指令取指时发生了存储器访问错误
    assign disp_o_alu_ilegl  = disp_i_ilegl  ; //该指令译码后发现其是一条非法指令
```

```
  // e203_exu_alu.v 源代码片段

  //Dispatch 模块和 ALU 之间的接口采用 valid-ready 模式的握手信号。
  // 对于源代码进行熟悉后读者可以发现，蜂鸟 E200 中的模块接口普遍采用这种握手接口。握手接口非
  // 常的简洁稳固。如果对于 valid-ready 模式的握手行为不甚理解的读者，可以参见第 12.2 节中
  // 对于 ICB 协议的详细介绍，描述了典型的 valid-ready 握手行为。
    input  i_valid,
    output i_ready,

  //ALU 内部将指令派遣给不同的 ALU 子单元

      // 对于发生取指异常的指令，单独列为一种类型，无须被具体的执行单元执行。
    wire ifu_excp_op = i_ilegl | i_buserr | i_misalgn;
    //通过 Decode 模块中译码生成的分组信息（包含在 Info Bus 中）进行判断，判别出需要什么
    //   单元执行此指令。ALU 主要包括 6 个功能子单元：
    //        普通 ALU 计算（Regular-ALU）：  主要负责普通的 ALU 指令
                                   （逻辑运算，加减法，移位等指令）执行
    //          访存地址生成（AGU）：主要负责 Load、Store 和“A”扩展指令的地址生成
    //        分支预测解析（BJP）：主要负责 Branch 和 Jump 指令的结果解析和执行
    //        CSR 读写控制（CSR-CTRL）：主要负责 CSR 指令的执行
    //        多周期乘除法器（MDV）：主要负责乘法和除法指令的执行
    wire alu_op = (~ifu_excp_op) & (i_info['E203_DECINFO_GRP] == 'E203_DECINF
O_GRP_ALU); //由 Regular-ALU 子单元执行
    wire agu_op = (~ifu_excp_op) & (i_info['E203_DECINFO_GRP] == 'E203_DECINF
O_GRP_AGU); //由 AGU 子单元执行
    wire bjp_op = (~ifu_excp_op) & (i_info['E203_DECINFO_GRP] == 'E203_DECINF
O_GRP_BJP); //由 BJP 子单元执行
    wire csr_op = (~ifu_excp_op) & (i_info['E203_DECINFO_GRP] == 'E203_DECINF
O_GRP_CSR); //由 CSR 子单元执行
   'ifdef E203_SUPPORT_SHARE_MULDIV //{
    wire mdv_op = (~ifu_excp_op) & (i_info['E203_DECINFO_GRP] == 'E203_DECINF
O_GRP_MULDIV); //由 MDV 子单元执行
   'endif//E203_SUPPORT_SHARE_MULDIV}

   // 根据不同的指令分组指示信号，将对应子单元的输入 valid 信号置高，并且选择对应子单元的 ready
   //  信号作为反馈给上游派遣模块的 ready 握手信号，通过此方式即实现了指令的派遣。
   //
   // 将对应子单元的输入 valid 信号置高。
   'ifdef E203_SUPPORT_SHARE_MULDIV //{
    wire mdv_i_valid = i_valid & mdv_op;
   'endif//E203_SUPPORT_SHARE_MULDIV}
    wire agu_i_valid = i_valid & agu_op;
    wire alu_i_valid = i_valid & alu_op;
```

```
    wire bjp_i_valid = i_valid & bjp_op;
    wire csr_i_valid = i_valid & csr_op;
    wire ifu_excp_i_valid = i_valid & ifu_excp_op;
  //
  //选择对应子单元的 ready 信号作为反馈给上游派遣模块的 ready 握手信号。本质上该逻辑是
  //使用 and-or 方式实现的并行多路选择器。
    assign i_ready =   (agu_i_ready & agu_op)
                    'ifdef E203_SUPPORT_SHARE_MULDIV //{
                    | (mdv_i_ready & mdv_op)
                    'endif//E203_SUPPORT_SHARE_MULDIV}
                    | (alu_i_ready & alu_op)
                    | (ifu_excp_i_ready & ifu_excp_op)
                    | (bjp_i_ready & bjp_op)
                    | (csr_i_ready & csr_op)
                     ;

  // 为了节省动态功耗，采用逻辑门控（Logic Gate）的方式，增加一级与门，对于子单元输入的信号
与分组指示
  //  信号进行“与”操作，那么在无须使用该子单元之时，其输入信号就都是 0，从而降低动态翻转功耗。
  //  请参见第 15 章了解更多低功耗设计的诀窍。
    wire  ['E203_XLEN-1:0]        csr_i_rs1   = {'E203_XLEN        {csr_op}} &
 i_rs1;
    wire  ['E203_XLEN-1:0]        csr_i_rs2   = {'E203_XLEN        {csr_op}} &
 i_rs2;
    wire  ['E203_XLEN-1:0]        csr_i_imm   = {'E203_XLEN        {csr_op}} &
 i_imm;
    wire  ['E203_DECINFO_WIDTH-1:0]  csr_i_info  = {'E203_DECINFO_WIDTH{csr_o
p}} & i_info;
    wire                          csr_i_rdwen =                     csr_op  &
 i_rdwen;

    wire  ['E203_XLEN-1:0]        bjp_i_rs1  = {'E203_XLEN         {bjp_op}} &
 i_rs1;
    wire  ['E203_XLEN-1:0]        bjp_i_rs2  = {'E203_XLEN         {bjp_op}} &
 i_rs2;
    wire  ['E203_XLEN-1:0]        bjp_i_imm  = {'E203_XLEN         {bjp_op}} &
 i_imm;
    wire  ['E203_DECINFO_WIDTH-1:0]  bjp_i_info = {'E203_DECINFO_WIDTH{bjp_op
}} & i_info;
    wire  ['E203_PC_SIZE-1:0]     bjp_i_pc   = {'E203_PC_SIZE      {bjp_op}} &
 i_pc;

    wire  ['E203_XLEN-1:0]        agu_i_rs1  = {'E203_XLEN         {agu_op}} &
 i_rs1;
    wire  ['E203_XLEN-1:0]        agu_i_rs2  = {'E203_XLEN         {agu_op}} &
 i_rs2;
    wire  ['E203_XLEN-1:0]        agu_i_imm  = {'E203_XLEN         {agu_op}} &
 i_imm;
    wire  ['E203_DECINFO_WIDTH-1:0] agu_i_info = {'E203_DECINFO_WIDTH{agu_op}} &
i_info;
    wire  ['E203_ITAG_WIDTH-1:0]  agu_i_itag = {'E203_ITAG_WIDTH   {agu_op}} &
 i_itag;
```

```
    wire  ['E203_XLEN-1:0]         alu_i_rs1  = {'E203_XLEN         {alu_op}} &
 i_rs1;
    wire  ['E203_XLEN-1:0]         alu_i_rs2  = {'E203_XLEN         {alu_op}} &
 i_rs2;
    wire  ['E203_XLEN-1:0]         alu_i_imm  = {'E203_XLEN         {alu_op}} &
 i_imm;
    wire  ['E203_DECINFO_WIDTH-1:0]  alu_i_info = {'E203_DECINFO_WIDTH{alu_op
}} & i_info;
    wire  ['E203_PC_SIZE-1:0]      alu_i_pc   = {'E203_PC_SIZE      {alu_op}} &
 i_pc;

    wire ['E203_XLEN-1:0]          mdv_i_rs1  = {'E203_XLEN         {mdv_op}} &
 i_rs1;
    wire ['E203_XLEN-1:0]          mdv_i_rs2  = {'E203_XLEN         {mdv_op}} &
 i_rs2;
    wire ['E203_XLEN-1:0]          mdv_i_imm  = {'E203_XLEN         {mdv_op}} &
 i_imm;
    wire ['E203_DECINFO_WIDTH-1:0] mdv_i_info = {'E203_DECINFO_WIDTH{mdv_op}} &
 i_info;
    wire  ['E203_ITAG_WIDTH-1:0] mdv_i_itag = {'E203_ITAG_WIDTH    {mdv_op}} &
 i_itag;
```

- 派遣模块中还会处理流水线冲突的问题，包括资源冲突和数据相关性造成的数据冲突。以下两节将专门予以论述。
- 派遣模块还会在某些特殊情况下将流水线的派遣点阻塞，相关源代码片段如下。

```
    // e203_exu_disp.v 源代码片段

  //派遣条件信号
   wire disp_condition =
                    //如果当前派遣指令需要访问 CSR 寄存器改变 CSR 的值，为了保险起见，必须等待
OITF 为空，也就意味着所有的长指令都已经执行完毕了，才会允许访问 CSR 寄存器的指令派遣从而改变 CSR
的值。下一节将介绍 OITF 和长指令的概念。
                    (disp_csr ? oitf_empty : 1'b1)
                    //如果当前派遣的指令属于 Fence 和 Fence.I 指令，同样必须等待 OITF 为空，也
就保证了 Fence 和 Fence.I 之前的指令都会被执行完毕。请参见附录 A14.2 节了解 Fence 和 Fence.I 指
令的详细信息。
                  & (disp_fence_fencei ? oitf_empty : 1'b1)
                    //如果已经交付了一条 WFI 指令，则必须立即阻塞派遣点，不让后续的指令派遣，
从而尽快让处理器进入 WFI 休眠状态，请参见附录 A14.2 节了解 WFI 指令的详细信息。
                  & (~wfi_halt_exu_req)
                    //如果发生了数据相关性，则阻塞派遣点。下一节将介绍数据相关性的检测。
                  & (~dep)
                    //如果当前派遣的是长指令，由于需要分配 OITF 表项，因此必须等待 OITF 有空。
下一节将介绍长指令的概念。
                  & (disp_alu_longp_prdt ? disp_oitf_ready : 1'b1);

  //只有满足派遣条件时，才会发生派遣
  assign disp_i_valid_pos = disp_condition & disp_i_valid;
  assign disp_i_ready     = disp_condition & disp_i_ready_pos;
```

8.3.7 流水线冲突、长指令和 OITF

1. 资源冲突

蜂鸟 E200 的执行阶段的一个最重要职能是维护并解决流水线的冲突，包括资源冲突和数据冲突（包括 WAW、WAR 和 RAW 等数据相关性）。

资源冲突通常发生在指令派遣给不同的执行单元进行执行的过程中。譬如，指令被派遣给除法单元进行运算，但是除法单元需要数十个周期才能完成此指令。那么后续的除法指令如果也需要被派遣给除法单元执行，便需要等待（出现了资源冲突）直到前一个指令完成操作，将除法单元释放出来。在蜂鸟 E200 的实现中，模块与模块的接口均采用严谨的 valid-ready 握手接口（如果对于 valid-ready 模式的握手行为不甚理解的读者，可以参见第 12.2 节中对于 ICB 协议的详细介绍，其描述了典型的 valid-ready 握手行为）。因此一旦某个模块当前不能够被使用（出现了资源冲突），那么它便会输出 ready 信号为低，从而无法完成握手。以 ALU 内部将指令派遣至各子单元为例，其源代码片段如下。

```
// e203_exu_alu.v 源代码片段

//
//选择派遣子单元的 ready 信号作为反馈给上游派遣模块的 ready 握手信号。
//   假设指令需要被派遣到 AGU 子单元执行，那么 agu_op 信号为高电平，选择 agu_i_ready 信号，
倘若此时 AGU 模块不能被使用（出现了资源冲突），那么它的 agu_i_ready 信号便会为低电平，从而使得
i_ready 信号为低电平，反馈给上游的派遣模块，无法完成握手，进而造成该指令无法被派遣，需要一直等
待直至 agu_i_ready 信号为高（资源冲突被解除）。
    assign i_ready =    (agu_i_ready & agu_op)
                      'ifdef E203_SUPPORT_SHARE_MULDIV //{
                      | (mdv_i_ready & mdv_op)
                      'endif//E203_SUPPORT_SHARE_MULDIV}
                      | (alu_i_ready & alu_op)
                      | (ifu_excp_i_ready & ifu_excp_op)
                      | (bjp_i_ready & bjp_op)
                      | (csr_i_ready & csr_op)
                      ;
```

2. 数据冲突

对于数据相关性引起的数据冲突，蜂鸟 E200 在执行阶段的处理比较巧妙。

首先，蜂鸟 E200 将所有需要执行的指令被分为两类。

- 一类是单周期执行指令。如图 8-2 所示，由于蜂鸟 E200 的交付功能和写回（Write-Back）功能均处于流水线的第二级，因此单周期执行指令在流水线的第二级便完成了交付，同时也将结果写回了 Regfile。
- 另一类是多周期执行的指令。这种指令通常需要多个周期才能够完成执行并写回，称之为“后交付长流水线指令（Post-Commit Write-Back Long-Pipes Instructions）”，

简称为 “长指令（Long-Pipes Instructions）”。

之所以如此命名，是因为多周期执行指令的写回操作要在多个周期后才能完成，而此指令的交付操作则已经在流水线的第二级就已经完成，因此写回和交付是在不同的周期内完成的，且写回是在交付完成之后，故称为 Post-Commit Write-Back。

蜂鸟 E200 是简单的按顺序单发射（派遣）微架构，在每条指令被进行派遣时，需要检查是否和之前派遣执行尚未写回的指令存在数据相关性。如第 6 章中所述，数据相关性分为 3 种。

（1）WAR（Write-After-Read）相关性

由于蜂鸟 E200 是按顺序派遣、按顺序写回的微架构，在指令的派遣时就已经从通用寄存器组中读取了源操作数。“后续执行的指令写回 Regfile 操作”不可能会发生在“前序执行的指令从 Regfile 中读取操作数”之前，因此不可能会发生 WAR 相关性造成的数据冲突。

（2）RAW（Read-After-Write）相关性

正在派遣的指令处于流水线的第二级，假设之前派遣的指令（简称前序指令）是单周期执行指令（也处于流水线的第二级写回），则前序指令肯定已经完成了执行并且将结果写回了 Regfile。因此正在派遣的指令不可能产生和前序单周期指令的 RAW 相关性造成的数据冲突。

但是假设之前派遣的指令（简称前序指令）是长指令，由于长指令需要多个周期才能写回结果，因此正在派遣的指令有可能产生与前序长指令的 RAW 相关性。

（3）WAW（Write-After-Write）相关性

正在派遣的指令处于流水线的第二级，假设之前派遣的指令（简称前序指令）是单周期执行指令（也处于流水线的第二级写回），则前序指令肯定已经完成了执行，并且将结果写回了 Regfile。因此正在派遣的指令不可能产生和前序单周期指令的 WAW 相关性造成的数据冲突。

但是假设之前派遣的指令（简称前序指令）是长指令，由于长指令需要多个周期才能写回结果，因此正在派遣的指令有可能产生与前序长指令的 WAW 相关性。

综上，在蜂鸟 E200 的流水线中，“正在派遣的指令”只可能与“尚未执行完毕的长指令”之间产生 RAW 和 WAW 相关性。

为了能够检测出与长指令的 RAW 和 WAW 相关性，蜂鸟 E200 使用了一个 Outstanding Instructions Track FIFO（OITF）模块，如图 8-4 所示。其要点如下。

- OITF 模块的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
     |----rtl                              // 存放 RTL 的目录
          |----e203                        // E203 核和 SoC 的 RTL 目录
               |----core                   // 存放 e203 Core 的 RTL 代码
                    |----e203_exu_oitf.v             // OITF 模块
```

- OITF 本质上是一个先入先出的 FIFO，FIFO 的深度默认为 2 个表项。

在流水线的派遣（Dispatch）点，每次派遣一个长指令，则会在 OITF 中分配一个表项（Entry），在这个表项中会存储该长指令的源操作数寄存器索引和结果寄存器索引。

在流水线的写回（Write-Back）点，每次按顺序写回一个长指令之后，就会将此指令在 OITF 中的表项去除，即从其 FIFO 退出（先入先出）完成其历史使命。如何保证长指令能够按顺序写回，也需要借助于 OITF 的功能。由于此功能阐述清楚需要较多篇幅，因此本书单设一章对其进行详述，请参见第 10 章了解有关写回（Write-Back）的详细介绍。因此 OITF 中存储的便是已经被派遣出且尚未写回的长指令信息。

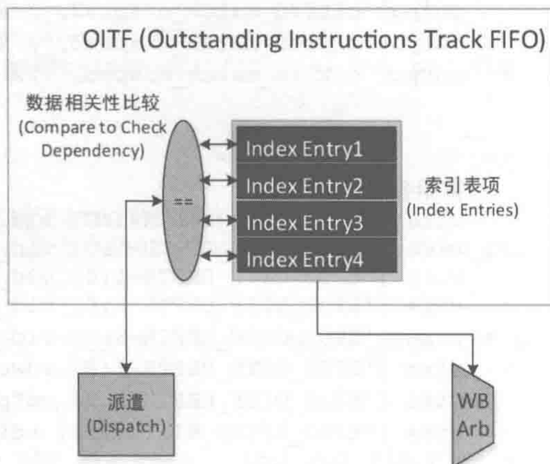


图 8-4　蜂鸟 E200 处理器核 OITF 微架构图

- 如图 8-4 所示，每条指令在派遣时，都会将本指令的源操作数寄存器索引和结果寄存器索引和 OITF 中的各个表项进行比对，从而判断本指令是否与已经被派遣出，且尚未写回长指令是否产生 RAW 和 WAW 相关性。
- 以上功能的 OITF 相关源代码片段如下。

```
// e203_exu_oitf.v 源代码片段

  input  dis_ena,//派遣一个长指令的使能信号，该信号将用于分配一个 OITF 表项
  input  ret_ena,//写回一个长指令的使能信号，该信号将用于移除一个 OITF 表项

 //以下为派遣的长指令相关信息，有的会被存储于 OITF 的表项中，有的会用于进行 RAW 和 WAW 判断
  input  disp_i_rs1en, //当前派遣指令是否需要读取第一个源操作数寄存器
  input  disp_i_rs2en, //当前派遣指令是否需要读取第二个源操作数寄存器
  input  disp_i_rs3en, //当前派遣指令是否需要读取第三个源操作数寄存器,
                       // 注意：只有浮点指令才会使用第三个源操作数
  input  disp_i_rdwen, //当前派遣指令是否需要写回结果寄存器
  input  disp_i_rs1fpu,    //当前派遣指令第一个源操作数是否是要读取浮点通用寄存器组
  input  disp_i_rs2fpu, //当前派遣指令第二个源操作数是否是要读取浮点通用寄存器组
  input  disp_i_rs3fpu, //当前派遣指令第三个源操作数是否是要读取浮点通用寄存器组
  input  disp_i_rdfpu, //当前派遣指令结果寄存器是否是要写回浮点通用寄存器组
  input  ['E203_RFIDX_WIDTH-1:0] disp_i_rs1idx, //当前派遣指令第一个源操作数寄存器
                                                 // 的索引
  input  ['E203_RFIDX_WIDTH-1:0] disp_i_rs2idx, //当前派遣指令第二个源操作数寄存器
                                                 // 的索引
  input  ['E203_RFIDX_WIDTH-1:0] disp_i_rs3idx, //当前派遣指令第三个源操作数寄存器
                                                 // 的索引
  input  ['E203_RFIDX_WIDTH-1:0] disp_i_rdidx,// 当前派遣指令的结果寄存器的索引
```

```
  input  ['E203_PC_SIZE   -1:0] disp_i_pc, //当前派遣指令的 PC

  output oitfrd_match_disprs1, //派遣指令源操作数一和 OITF 任一表项中的结果寄存器相同
  output oitfrd_match_disprs2, //派遣指令源操作数二和 OITF 任一表项中的结果寄存器相同
  output oitfrd_match_disprs3, //派遣指令源操作数三和 OITF 任一表项中的结果寄存器相同
  output oitfrd_match_disprd, //派遣指令结果寄存器和 OITF 任一表项中的结果寄存器相同

  ......

 //声明各表项的信号
  wire ['E203_OITF_DEPTH-1:0] vld_set;
  wire ['E203_OITF_DEPTH-1:0] vld_clr;
  wire ['E203_OITF_DEPTH-1:0] vld_ena;
  wire ['E203_OITF_DEPTH-1:0] vld_nxt;
  wire ['E203_OITF_DEPTH-1:0] vld_r;    //各表项中是否存放了有效指令的指示信号
  wire ['E203_OITF_DEPTH-1:0] rdwen_r;  //各表项中指令是否写回结果寄存器
  wire ['E203_OITF_DEPTH-1:0] rdfpu_r;  //各表项中指令写回的结果寄存器是否属于浮点
  wire ['E203_RFIDX_WIDTH-1:0] rdidx_r['E203_OITF_DEPTH-1:0];//各表项中指令的结
                                                              //   果寄存器索引
  // The PC here is to be used at wback stage to track out the
  //  PC of exception of long-pipe instruction
  wire ['E203_PC_SIZE-1:0] pc_r['E203_OITF_DEPTH-1:0]; //各表项中指令的 PC

 //由于 OITF 本质上是一个 FIFO，因此需要生成 FIFO 的写指针。

  wire alc_ptr_ena = dis_ena;//派遣一个长指令的使能信号，作为写指针的使能信号
  wire ret_ptr_ena = ret_ena;//写回一个长指令的使能信号，作为读指针的使能信号

  wire oitf_full ;

  wire ['E203_ITAG_WIDTH-1:0] alc_ptr_r;
  wire ['E203_ITAG_WIDTH-1:0] ret_ptr_r;

    //与常规的 FIFO 设计一样，为了方便维护空满标志，为写指针增加额外的一个标志位
      wire alc_ptr_flg_r;
      wire alc_ptr_flg_nxt = ~alc_ptr_flg_r;
      wire alc_ptr_flg_ena = (alc_ptr_r == ($unsigned('E203_OITF_DEPTH-1)))
 & alc_ptr_ena;

      sirv_gnrl_dfflr #(1) alc_ptr_flg_dfflrs(alc_ptr_flg_ena, alc_ptr_flg_
nxt, alc_ptr_flg_r, clk, rst_n);

      wire ['E203_ITAG_WIDTH-1:0] alc_ptr_nxt;
      //每次分配一个表项，写指针自增 1，如果达到了 FIFO 的深度值，写指针归零
      assign alc_ptr_nxt = alc_ptr_flg_ena ? 'E203_ITAG_WIDTH'b0 : (alc_ptr
_r + 1'b1);

      sirv_gnrl_dfflr #('E203_ITAG_WIDTH) alc_ptr_dfflrs(alc_ptr_ena, alc_p
tr_nxt, alc_ptr_r, clk, rst_n);

    //与常规的 FIFO 设计一样，为了方便维护空满标志，为读指针增加额外的一个标志位
```

```
        wire ret_ptr_flg_r;
        wire ret_ptr_flg_nxt = ~ret_ptr_flg_r;
        wire ret_ptr_flg_ena = (ret_ptr_r == ($unsigned('E203_OITF_DEPTH-1)))
 & ret_ptr_ena;

        sirv_gnrl_dfflr #(1) ret_ptr_flg_dfflrs(ret_ptr_flg_ena, ret_ptr_flg_
nxt, ret_ptr_flg_r, clk, rst_n);

        wire ['E203_ITAG_WIDTH-1:0] ret_ptr_nxt;

        //每次移除一个表项，读指针自增1，如果达到了FIFO的深度值，读指针归零
        assign ret_ptr_nxt = ret_ptr_flg_ena ? 'E203_ITAG_WIDTH'b0 : (ret_ptr
_r + 1'b1);

        sirv_gnrl_dfflr #('E203_ITAG_WIDTH) ret_ptr_dfflrs(ret_ptr_ena, ret_p
tr_nxt, ret_ptr_r, clk, rst_n);

      //生成FIFO的空满标志
        assign oitf_empty = (ret_ptr_r == alc_ptr_r) &   (ret_ptr_flg_r == al
c_ptr_flg_r);
        assign oitf_full  = (ret_ptr_r == alc_ptr_r) & (~(ret_ptr_flg_r == al
c_ptr_flg_r));
    ……


     wire ['E203_OITF_DEPTH-1:0] rd_match_rs1idx;
     wire ['E203_OITF_DEPTH-1:0] rd_match_rs2idx;
     wire ['E203_OITF_DEPTH-1:0] rd_match_rs3idx;
     wire ['E203_OITF_DEPTH-1:0] rd_match_rdidx;

     genvar i;
     generate //{//使用参数化的generate语法实现FIFO的主体部分
        for (i=0; i<'E203_OITF_DEPTH; i=i+1) begin:oitf_entries//{

   //生成各表项中是否存放了有效指令的指示信号
            //每次分配一个表项时，且写指针与当前表项编号一样，则将该表项的有效信号设置为高
          assign vld_set[i] = alc_ptr_ena & (alc_ptr_r == i);
            //每次移除一个表项时，且读指针与当前表项编号一样，则将该表项的有效信号清除为低
          assign vld_clr[i] = ret_ptr_ena & (ret_ptr_r == i);
          assign vld_ena[i] = vld_set[i] |   vld_clr[i];
          assign vld_nxt[i] = vld_set[i] | (~vld_clr[i]);

          sirv_gnrl_dfflr #(1) vld_dfflrs(vld_ena[i], vld_nxt[i], vld_r[i], c
lk, rst_n);
            //其他的表项信息，均可视为该表项的载荷（Payload），只需要在表项分配时写入，在
            //表项移除时无需清除（为了节省动态功耗，请参见第15章了解更多低功耗设计的诀窍）

          sirv_gnrl_dffl #('E203_RFIDX_WIDTH) rdidx_dfflrs(vld_set[i], disp_i
_rdidx, rdidx_r[i], clk);//各表项中指令的结果寄存器索引
          sirv_gnrl_dffl #('E203_PC_SIZE    ) pc_dfflrs    (vld_set[i], disp_i
```

```
_pc    , pc_r[i]    , clk);// 各表项中指令的 PC
          sirv_gnrl_dffl #(1)             rdwen_dfflrs(vld_set[i], disp_i
_rdwen, rdwen_r[i], clk); //各表项中指令是否需要写回结果寄存器
          sirv_gnrl_dffl #(1)             rdfpu_dfflrs(vld_set[i], disp_i
_rdfpu, rdfpu_r[i], clk);//各表项中指令写回的结果寄存器是否属于浮点

   //将正在派遣的指令的源操作数寄存器索引和各表项中的结果寄存器索引进行比较
        assign rd_match_rs1idx[i] = vld_r[i] & rdwen_r[i] & disp_i_rs1en & (r
dfpu_r[i] == disp_i_rs1fpu) & (rdidx_r[i] == disp_i_rs1idx);
        assign rd_match_rs2idx[i] = vld_r[i] & rdwen_r[i] & disp_i_rs2en & (r
dfpu_r[i] == disp_i_rs2fpu) & (rdidx_r[i] == disp_i_rs2idx);
        assign rd_match_rs3idx[i] = vld_r[i] & rdwen_r[i] & disp_i_rs3en & (r
dfpu_r[i] == disp_i_rs3fpu) & (rdidx_r[i] == disp_i_rs3idx);
    //将正在派遣的指令的结果寄存器索引和各表项中的结果寄存器索引进行比较
       assign rd_match_rdidx [i] = vld_r[i] & rdwen_r[i] & disp_i_rdwen & (rd
fpu_r[i] == disp_i_rdfpu ) & (rdidx_r[i] == disp_i_rdidx );

       end//}
    endgenerate//}

   //派遣指令源操作数一和 OITF 任一表项中的结果寄存器相同，表示存在着 RAW 相关性
     assign oitfrd_match_disprs1 = |rd_match_rs1idx;
   //派遣指令源操作数二和 OITF 任一表项中的结果寄存器相同，表示存在着 RAW 相关性
     assign oitfrd_match_disprs2 = |rd_match_rs2idx;
   //派遣指令源操作数三和 OITF 任一表项中的结果寄存器相同，表示存在着 RAW 相关性
     assign oitfrd_match_disprs3 = |rd_match_rs3idx;
   //派遣指令结果寄存器和 OITF 任一表项中的结果寄存器相同，表示存在着 WAW 相关性
     assign oitfrd_match_disprd  = |rd_match_rdidx ;
```

- 在流水线的派遣点，每条指令在派遣之时如果发现了数据相关性，则会将流水线的派遣点阻塞，直到相关长指令执行完毕解除了相关性之后才会继续进行派遣。此功能由 Dispatch 模块完成，相关性的源代码片段如下：

```
   // e203_exu_disp.v 源代码片段

  //只要源操作数 1、2、3 中的任何一个和 OITF 中的表项产生了 RAW 相关性，则意味着该指令和前序的
长指令存在着 RAW 相关性
   wire raw_dep =  ((oitfrd_match_disprs1) |
                    (oitfrd_match_disprs2) |
                    (oitfrd_match_disprs3));
  //只要结果寄存器和 OITF 中的表项产生了 WAW 相关性，则意味着该指令和和前序的长指令存在着 WAW
相关性
   wire waw_dep = (oitfrd_match_disprd);

   // RAW 和 WAW 两种相关性都是需要阻塞派遣点的相关性
   wire dep = raw_dep | waw_dep;

    wire disp_condition =
                   ......
                 & (~dep)    //没发生 RAW 和 WAW 相关性时才会允许派遣
```

```
……
assign disp_i_valid_pos = disp_condition & disp_i_valid;
assign disp_i_ready     = disp_condition & disp_i_ready_pos;
```

从以上介绍可以看出，蜂鸟 E200 对于数据相关性造成的冲突，只采取了阻塞流水线的方法，而并没有将长指令的结果直接快速旁路给后续的待派遣指令。使用该方案主要是因为处理器的设计需要秉承折中取舍的哲学，由于蜂鸟 E200 主要侧重于低功耗和小面积，因此没有使用快速旁路的方法。

8.3.8 ALU

蜂鸟 E200 的 ALU 单元，包括 5 个功能子单元，各自的功能分为以下 5 种情况。

- 普通 ALU 运算：主要负责普通的 ALU 指令（逻辑运算，加减法，移位等指令）的执行。
- 访存地址生成：主要负责 Load、Store 和 “A” 扩展指令的地址生成。
- 分支预测解析：主要负责 Branch 和 Jump 指令的结果解析和执行。
- CSR 读写控制：主要负责 CSR 指令的执行。
- 多周期乘除法器：主要负责乘法和除法指令的执行。

如图 8-5 所示，以上 5 个功能子单元只负责具体指令执行的控制，它们均共享一份实际的运算数据通路，因此主要数据通路的面积开销只有一份，这也是蜂鸟 E200 追求低功耗小面积实现的亮点。

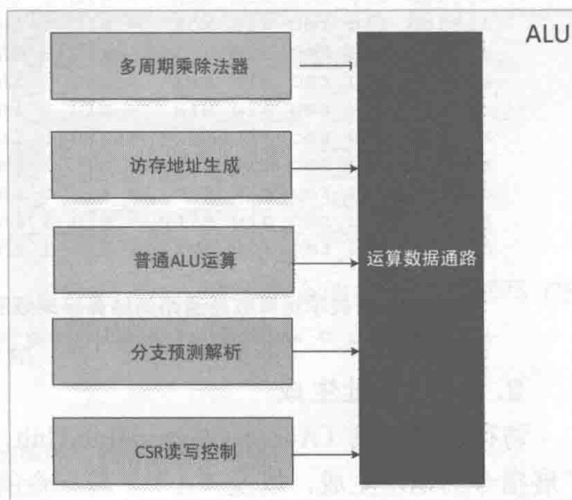


图 8-5 蜂鸟 E200 处理器核的 ALU 框图

1. 普通 ALU 运算

普通 ALU 运算（Regular-ALU）模块，主要负责普通的 ALU 指令（逻辑运算，加减法和移位等指令）的执行，其要点如下。

- Regular-ALU 模块的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                              // 存放 RTL 的目录
        |----e203                         // E203 核和 SoC 的 RTL 目录
            |----core                     // 存放 e203 Core 的 RTL 代码
                |----e203_exu_alu_rglr.v          // Regular-ALU 运算模块
```

- Regular-ALU 模块完全由组合逻辑组成，Regular-ALU 模块本身并没有运算数据通路，其主要逻辑即根据普通 ALU 指令类型，发起对共享运算数据通路的操作请求，并且从共享的运算数据通路中取回计算结果。相关源代码片段如下所示。

```
// e203_exu_alu_rglr.v 源代码片段

//从 Info Bus 中取出相关信息

       //本指令的第二个源操作数是否使用立即数
  wire op2imm  = alu_i_info ['E203_DECINFO_ALU_OP2IMM ];
       //本指令的第一个源操作数是否使用 PC
  wire op1pc   = alu_i_info ['E203_DECINFO_ALU_OP1PC  ];
     //将操作数 1 发送给共享的运算数据通路，如果使用 PC 则选择 PC，否则选择源寄存器 1
  assign alu_req_alu_op1  = op1pc  ? alu_i_pc  : alu_i_rs1;
     //将操作数 2 发送给共享的运算数据通路，如果使用立即数则选择立即数，否则选择源寄存器 2
  assign alu_req_alu_op2  = op2imm ? alu_i_imm : alu_i_rs2;
      // 根据指令的类型，产生所需计算的操作类型，将其发送给共享的运算数据通路
  assign alu_req_alu_add  = alu_i_info ['E203_DECINFO_ALU_ADD ] & (~nop);
  assign alu_req_alu_sub  = alu_i_info ['E203_DECINFO_ALU_SUB ];
  assign alu_req_alu_xor  = alu_i_info ['E203_DECINFO_ALU_XOR ];
  assign alu_req_alu_sll  = alu_i_info ['E203_DECINFO_ALU_SLL ];
  assign alu_req_alu_srl  = alu_i_info ['E203_DECINFO_ALU_SRL ];
  assign alu_req_alu_sra  = alu_i_info ['E203_DECINFO_ALU_SRA ];
  assign alu_req_alu_or   = alu_i_info ['E203_DECINFO_ALU_OR  ];
  assign alu_req_alu_and  = alu_i_info ['E203_DECINFO_ALU_AND ];
  assign alu_req_alu_slt  = alu_i_info ['E203_DECINFO_ALU_SLT ];
  assign alu_req_alu_sltu = alu_i_info ['E203_DECINFO_ALU_SLTU];
  assign alu_req_alu_lui  = alu_i_info ['E203_DECINFO_ALU_LUI ];

        //将共享运算数据通路的结算结果取回
  assign alu_o_wbck_wdat = alu_req_alu_res;
```

2. 访存地址生成

访存地址生成（Address Generation Unit，AGU）模块，主要负责 Load、Store 和“A”扩展指令的地址生成，以及“A”扩展指令的微操作拆分和执行。AGU 模块相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
      |----rtl                              // 存放 RTL 的目录
           |----e203                        // E203 核和 SoC 的 RTL 目录
                |----core                   // 存放 e203 Core 的 RTL 代码
                     |----e203_exu_alu_lsuagu.v        // AGU 模块
```

AGU 模块由于是整个存储器访问指令执行的一个小环节，需要结合完整的存储器访问微架构进行理解。在第 11.4.2 节中有对 AGU 模块的详细解析，本章在此不做赘述，请读者参见第 11 章了解更多信息。

3. 分支预测解析

分支预测解析（Branch and Jump resolve，BJP）模块，主要负责 Branch 和 Jump 指令的结果解析和执行。BJP 相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                    // 存放 RTL 的目录
        |----e203                               // E203 核和 SoC 的 RTL 目录
            |----core                           // 存放 e203 Core 的 RTL 代码
                |----e203_exu_alu_bjp.v              // BJP 模块
```

BJP 模块是分支跳转指令进行交付的主要依据，在第 9.3.1 节中有对 BJP 模块的详细解析，本章在此不做赘述，请读者参见第 9 章了解更多信息。

4. CSR 读写控制

CSR 读写控制（CSR-CTRL）模块，主要负责 CSR 读写指令的执行，包括 CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI 以及 CSRRCI 指令，其要点如下。

- CSR-CTRL 模块的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                    // 存放 RTL 的目录
        |----e203                               // E203 核和 SoC 的 RTL 目录
            |----core                           // 存放 e203 Core 的 RTL 代码
                |----e203_exu_alu_csrctrl.v              // CSR-CTRL 模块
```

- CSR-CTRL 模块完全由组合逻辑组成，其根据 CSR 读写指令的类型产生读写 CSR 寄存器模块的控制信号。相关源代码片段如下所示。

```
// e203_exu_alu_csrctrl.v 源代码片段

  output csr_ena,         //通给 CSR 寄存器模块的 CSR 读写使能信号
  output csr_wr_en,       //CSR 写操作指示信号
  output csr_rd_en,       //CSR 读操作指示信号
  output [12-1:0] csr_idx,//CSR 寄存器的地址索引

  input  ['E203_XLEN-1:0] read_csr_dat,//读操作从 CSR 寄存器模块中读出的数据
  output ['E203_XLEN-1:0] wbck_csr_dat,//写操作写入 CSR 寄存器模块的数据

......

//从 Info Bus 中取出相关信息
  wire         csrrw  = csr_i_info['E203_DECINFO_CSR_CSRRW ];
  wire         csrrs  = csr_i_info['E203_DECINFO_CSR_CSRRS ];
  wire         csrrc  = csr_i_info['E203_DECINFO_CSR_CSRRC ];
  wire         rs1imm = csr_i_info['E203_DECINFO_CSR_RS1IMM];
  wire         rs1is0 = csr_i_info['E203_DECINFO_CSR_RS1IS0];
```

```
    wire [4:0]  zimm   = csr_i_info['E203_DECINFO_CSR_ZIMMM ];
    wire [11:0] csridx = csr_i_info['E203_DECINFO_CSR_CSRIDX];

      //生成操作数 1，如果使用立即数则选择立即数，否则选择源寄存器 1
    wire ['E203_XLEN-1:0] csr_op1 = rs1imm ? {27'b0,zimm} : csr_i_rs1;

      //根据指令的信息生成读操作指示信号
    assign csr_rd_en = csr_i_valid &
      (
        (csrrw ? csr_i_rdwen : 1'b0) // the CSRRW only read when the destinat
ion reg need to be writen
        | csrrs | csrrc // The set and clear operation always need to read CSR
      );

      //根据指令的信息生成写操作指示信号
    assign csr_wr_en = csr_i_valid & (
                csrrw // CSRRW always write the original RS1 value into the CSR
              | ((csrrs | csrrc) & (~rs1is0)) // for CSRRS/RC, if the RS is x
0, then should not really write
              );

      //生成访问 CSR 寄存器的地址索引
    assign csr_idx = csridx;

      //生成通给 CSR 寄存器模块的 CSR 读写使能信号
    assign csr_ena = csr_o_valid & csr_o_ready;

      //生成写操作写入 CSR 寄存器模块的数据
    assign wbck_csr_dat =
                ({'E203_XLEN{csrrw}} & csr_op1)
              | ({'E203_XLEN{csrrs}} & (  csr_op1  | read_csr_dat))
              | ({'E203_XLEN{csrrc}} & ((~csr_op1) & read_csr_dat));
```

5. 多周期乘除法器

蜂鸟 E200 系列对于乘除法指令的支持有两种不同的配置。一种配置是高性能的单周期乘法器和独立的除法器，此配置会在第 8.3.9 节介绍。另一种配置是使用低性能小面积的多周期乘除法器。

对于多周期的乘法和除法器实现，先对其知识背景进行简介如下。

- 对于有符号整数乘法操作，可以使用常用的 Booth 编码产生部分积，然后使用迭代的方法，每个周期使用加法器对部分积进行累加，经过多个周期的迭代之后得到最终的乘积。其基本硬件原理图如图 8-6 所示，从而实现多周期乘法器。
  本书在此仅对乘法器实现进行简介。对于使用 Booth 编码算法进行乘法器设计的详细原理，本书不做赘述，请读者自行查阅相关资料学习。
- 对于有符号整数除法操作，可以使用常用的加减交替法，然后使用迭代的方法，每

个周期使用加法器生成部分余数，经过多个周期的迭代之后得到最终商和余数。其基本硬件原理图如图 8-7 所示，从而实现多周期除法器。

本书在此仅对除法器实现进行简介。对于使用加减交替法进行除法器设计的详细原理，本书在此不做赘述，请读者自行查阅相关资料学习。

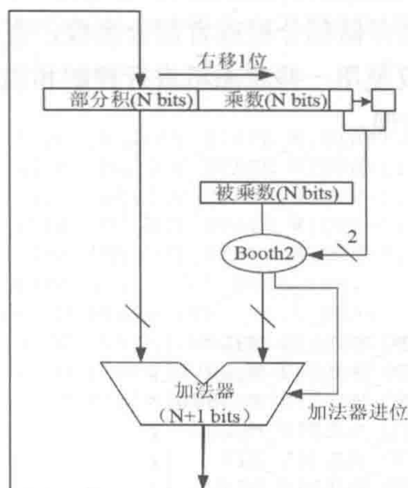


图 8-6 使用加法器进行迭代的乘法器实现

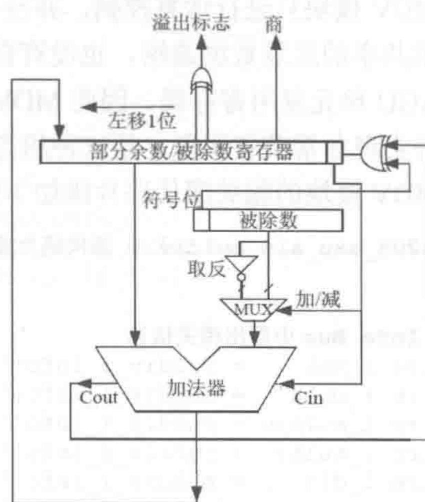


图 8-7 使用加法器进行迭代的除法器实现

通过对多周期乘法器和多周期除法器的实现进行比较可以发现，其结构非常类似，二者同样使用加法器作为主要的运算通路，使用一组寄存器保存部分积或者部分余数，因此二者可以进行资源复用，从而节省面积开销。

根据上述的设计思路，在蜂鸟 E200 中，多周期乘除法器（MDV）模块是 ALU 的一个子单元，通过复用 ALU 共享数据通路中的加法器，经过多个周期完成乘法或者除法操作，其要点如下。

- MDV 模块的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                              // 存放 RTL 的目录
        |----e203                         // E203 核和 SoC 的 RTL 目录
            |----core                     // 存放 e203 Core 的 RTL 代码
                |----e203_exu_alu_muldiv.v        // 多周期乘除法单元
```

- 对于乘法操作，为了减少乘法操作所需的周期数，MDV 对乘法采用基 4（Radix-4）的 Booth 编码，并且对于无符号乘法进行一位符号扩展后统一当作有符号数进行运算，因此需要 17 个迭代周期。
- 对于除法操作，采用普通的加减交替法，同样对于无符号乘法进行一位符号扩展后

统一当作有符号数进行运算，因此需要33个迭代周期。另外，由于加减交替法迭代所得的结果存在着1比特精度的问题，因此还需要额外1个周期判断是否需要进行商和余数的矫正，和额外2个周期的商和余数矫正（如果需要矫正），最终得到完全准确的除法结果，总共最多需要36个周期。

- MDV模块只进行运算控制，并没有自己的加法器，加法器与其他的ALU子单元复用共享的运算数据通路，也没有自己的寄存器存储部分积或者部分余数，寄存器与AGU单元复用寄存器。因此MDV单元本身仅使用一些状态机进行控制和选择，硬件实现非常节省面积，是一种相当低功耗的实现。
- MDV模块的相关源代码片段如下所示。

```
// e203_exu_alu_muldiv.v 源代码片段


//从 Info Bus 中取出相关信息
  wire i_mul    = muldiv_i_info['E203_DECINFO_MULDIV_MUL   ];
  wire i_mulh   = muldiv_i_info['E203_DECINFO_MULDIV_MULH  ];
  wire i_mulhsu = muldiv_i_info['E203_DECINFO_MULDIV_MULHSU];
  wire i_mulhu  = muldiv_i_info['E203_DECINFO_MULDIV_MULHU ];
  wire i_div    = muldiv_i_info['E203_DECINFO_MULDIV_DIV   ];
  wire i_divu   = muldiv_i_info['E203_DECINFO_MULDIV_DIVU  ];
  wire i_rem    = muldiv_i_info['E203_DECINFO_MULDIV_REM   ];
  wire i_remu   = muldiv_i_info['E203_DECINFO_MULDIV_REMU  ];

//对操作数进行符号位扩展
  wire mul_rs1_sign = (i_mulhu)           ? 1'b0 : muldiv_i_rs1['E203_XLEN-1];
  wire mul_rs2_sign = (i_mulhsu | i_mulhu) ? 1'b0 : muldiv_i_rs2['E203_XLEN-1];

  wire [32:0] mul_op1 = {mul_rs1_sign, muldiv_i_rs1};
  wire [32:0] mul_op2 = {mul_rs2_sign, muldiv_i_rs2};
//译码出乘法还是除法操作
  wire i_op_mul = i_mul | i_mulh | i_mulhsu | i_mulhu;
  wire i_op_div = i_div | i_divu | i_rem | i_remu;

//使用统一的状态机控制多周期乘法或者除法操作
  ////////////////////////////////////////////////////////////////////////////
////////
  // Implement the state machine for
  //    (1) The MUL instructions
  //    (2) The DIV instructions
  localparam MULDIV_STATE_WIDTH = 3;

  wire [MULDIV_STATE_WIDTH-1:0] muldiv_state_nxt;
  wire [MULDIV_STATE_WIDTH-1:0] muldiv_state_r;
  wire muldiv_state_ena;

  // State 0: The 0th state, means this is the 1 cycle see the operand inputs
```

```
    localparam MULDIV_STATE_0TH = 3'd0;
    // State 1: Executing the instructions
    localparam MULDIV_STATE_EXEC = 3'd1;
    // State 2: Div check if need correction
    localparam MULDIV_STATE_REMD_CHCK = 3'd2;
    // State 3: Quotient correction
    localparam MULDIV_STATE_QUOT_CORR = 3'd3;
    // State 4: Reminder correction
    localparam MULDIV_STATE_REMD_CORR = 3'd4;
    ......

    wire [MULDIV_STATE_WIDTH-1:0] state_0th_nxt;
    wire [MULDIV_STATE_WIDTH-1:0] state_exec_nxt;
    wire [MULDIV_STATE_WIDTH-1:0] state_remd_chck_nxt;
    wire [MULDIV_STATE_WIDTH-1:0] state_quot_corr_nxt;
    wire [MULDIV_STATE_WIDTH-1:0] state_remd_corr_nxt;
    wire state_0th_exit_ena;
    wire state_exec_exit_ena;
    wire state_remd_chck_exit_ena;
    wire state_quot_corr_exit_ena;
    wire state_remd_corr_exit_ena;

    ......

      sirv_gnrl_dfflr #(MULDIV_STATE_WIDTH) muldiv_state_dfflr (muldiv_state_
ena, muldiv_state_nxt, muldiv_state_r, clk, rst_n);

    wire state_exec_enter_ena = muldiv_state_ena & (muldiv_state_nxt == MULDI
V_STATE_EXEC);


    localparam EXEC_CNT_W  = 6;
    localparam EXEC_CNT_1  = 6'd1 ;
    localparam EXEC_CNT_16 = 6'd16; //指示乘法需要总共17个迭代时钟周期
    localparam EXEC_CNT_32 = 6'd32; //指示乘法需要总共33个迭代时钟周期

  //实现迭代周期的计数
    wire[EXEC_CNT_W-1:0] exec_cnt_r;
    wire exec_cnt_set = state_exec_enter_ena;
    wire exec_cnt_inc = muldiv_sta_is_exec & (~exec_last_cycle);
    wire exec_cnt_ena = exec_cnt_inc | exec_cnt_set;
      // When set, the counter is set to 1, because the 0th state also counte
d as 0th cycle
    wire[EXEC_CNT_W-1:0] exec_cnt_nxt = exec_cnt_set ? EXEC_CNT_1 : (exec_cnt
_r + 1'b1);
    sirv_gnrl_dfflr #(EXEC_CNT_W) exec_cnt_dfflr (exec_cnt_ena, exec_cnt_nxt,
 exec_cnt_r, clk, rst_n);

  ......

  //使用基4编码算法生成乘法的部分积
  ......
```

```
    wire [2:0] booth_code = cycle_0th  ? {muldiv_i_rs1[1:0],1'b0}
              : cycle_16th ? {mul_rs1_sign,part_prdt_lo_r[0],part_prdt_sft1_r}
                      : {part_prdt_lo_r[1:0],part_prdt_sft1_r};
        //booth_code == 3'b000 =  0
        //booth_code == 3'b001 =  1
        //booth_code == 3'b010 =  1
        //booth_code == 3'b011 =  2
        //booth_code == 3'b100 = -2
        //booth_code == 3'b101 = -1
        //booth_code == 3'b110 = -1
        //booth_code == 3'b111 = -0
    wire booth_sel_zero = (booth_code == 3'b000) | (booth_code == 3'b111);
    wire booth_sel_two  = (booth_code == 3'b011) | (booth_code == 3'b100);
    wire booth_sel_one  = (~booth_sel_zero) & (~booth_sel_two);
    wire booth_sel_sub  = booth_code[2];

     //生成乘法每次迭代所需加法或者减法的操作数，使用 and-or 的编码方式产生并行选择器
    wire ['E203_MULDIV_ADDER_WIDTH-1:0] mul_exe_alu_op2 =
        ({'E203_MULDIV_ADDER_WIDTH{booth_sel_zero}} & 'E203_MULDIV_ADDER_WIDT
H'b0)
      | ({'E203_MULDIV_ADDER_WIDTH{booth_sel_one }} & {mul_rs2_sign,mul_rs2_s
ign,mul_rs2_sign,muldiv_i_rs2})
      | ({'E203_MULDIV_ADDER_WIDTH{booth_sel_two }} & {mul_rs2_sign,mul_rs2_s
ign,muldiv_i_rs2,1'b0})
        ;
    wire ['E203_MULDIV_ADDER_WIDTH-1:0] mul_exe_alu_op1 =
         cycle_0th ? 'E203_MULDIV_ADDER_WIDTH'b0 : {part_prdt_hi_r[32],part_p
rdt_hi_r[32],part_prdt_hi_r};

     ……

     //生成乘法每次迭代所需进行加法或者减法的指示信号
    wire mul_exe_alu_add = (~booth_sel_sub);
    wire mul_exe_alu_sub = booth_sel_sub;

  ……

  //使用加减交替法生成除法的部分余数和商

  ……
     //生成除法每次迭代所需加法或者减法的操作数
    wire [33:0] div_exe_alu_op1= cycle_0th ? dividend_lsft1[66:33] : {part_r
emd_sft1_r, part_remd_r[32:0]};
    wire [33:0] div_exe_alu_op2 = divisor;
    wire div_exe_alu_add = (~prev_quot);
    wire div_exe_alu_sub =  prev_quot ;

   ……

  //判定是否需要进行商和余数的矫正，此判定需要用到加法器，将操作数发送给共享的数据通路
    wire [33:0] div_remd_chck_alu_res = muldiv_req_alu_res[33:0];
    wire [33:0] div_remd_chck_alu_op1 = {part_remd_r[32], part_remd_r};
```

```
  wire [33:0] div_remd_chck_alu_op2 = divisor;
  wire div_remd_chck_alu_add = 1'b1;
  wire div_remd_chck_alu_sub = 1'b0;

  wire remd_is_0 = ~(|part_remd_r);
  wire remd_is_neg_divs = ~(|div_remd_chck_alu_res);
  wire remd_is_divs = (part_remd_r == divisor[32:0]);
  assign div_need_corrct = i_op_div & (
                         ((part_remd_r[32] ^ dividend[65]) & (~remd_is_0))
                       | remd_is_neg_divs
                       | remd_is_divs
                     );

  wire remd_inc_quot_dec = (part_remd_r[32] ^ divisor[33]);

......

//进行商的矫正所需的加法运算，将操作数和操作类型发送给共享的数据通路
  assign div_quot_corr_alu_res = muldiv_req_alu_res[33:0];
  wire [33:0] div_quot_corr_alu_op1 = {part_quot_r[32], part_quot_r};
  wire [33:0] div_quot_corr_alu_op2 = 34'b1;
  wire div_quot_corr_alu_add = (~remd_inc_quot_dec);
  wire div_quot_corr_alu_sub = remd_inc_quot_dec;
......

//进行余数矫正所需的加法运算，将操作数和操作类型发送给共享的数据通路
  assign div_remd_corr_alu_res = muldiv_req_alu_res[33:0];
  wire [33:0] div_remd_corr_alu_op1 = {part_remd_r[32], part_remd_r};
  wire [33:0] div_remd_corr_alu_op2 = divisor;
  wire div_remd_corr_alu_add = remd_inc_quot_dec;
  wire div_remd_corr_alu_sub = ~remd_inc_quot_dec;

......

//为了与 ALU 的其他子单元共享运算数据通路，将运算所需要的操作数发送给运算数据通路进行运算
    //操作数 1
  assign muldiv_req_alu_op1 =
                  ......
    //操作数 2
  assign muldiv_req_alu_op2 =
                 ......
    //指示需要进行加法操作
  assign muldiv_req_alu_add  =
           (req_alu_sel1 & mul_exe_alu_add       )
         | (req_alu_sel2 & div_exe_alu_add       )
         | (req_alu_sel3 & div_quot_corr_alu_add)
         | (req_alu_sel4 & div_remd_corr_alu_add)
         | (req_alu_sel5 & div_remd_chck_alu_add);
    //指示需要进行减法操作
  assign muldiv_req_alu_sub  =
```

```
            (req_alu_sel1 & mul_exe_alu_sub        )
          | (req_alu_sel2 & div_exe_alu_sub        )
          | (req_alu_sel3 & div_quot_corr_alu_sub)
          | (req_alu_sel4 & div_remd_corr_alu_sub)
          | (req_alu_sel5 & div_remd_chck_alu_sub);

//为了与 AGU 单元共享运算数据通路，将需要存储的部分积或者部分余数发送给运算数据通路进行寄存
  assign muldiv_sbf_0_ena = part_remd_ena | part_prdt_hi_ena;
  assign muldiv_sbf_0_nxt = i_op_mul ? part_prdt_hi_nxt : part_remd_nxt;

  assign muldiv_sbf_1_ena = part_quot_ena | part_prdt_lo_ena;
  assign muldiv_sbf_1_nxt = i_op_mul ? part_prdt_lo_nxt : part_quot_nxt;

  assign part_remd_r = muldiv_sbf_0_r;
  assign part_quot_r = muldiv_sbf_1_r;
  assign part_prdt_hi_r = muldiv_sbf_0_r;
  assign part_prdt_lo_r = muldiv_sbf_1_r;

......
```

6. 运算数据通路

运算数据通路模块是 ALU 真正用于计算的数据通路模块。运算数据通路模块的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
   |----rtl                              // 存放 RTL 的目录
        |----e203                        // E203 核和 SoC 的 RTL 目录
             |----core                   // 存放 e203 Core 的 RTL 代码
                  |----e203_exu_alu_dpath.v      // 运算数据通路模块
```

运算数据通路的功能比较简单，它被动地接受其他 ALU 子单元的请求进行具体运算，然后将计算结果返回给其他子单元运算数据通路，相关源代码片段如下所示。

```
    // e203_exu_alu_dpath.v 源代码片段

    ......

    //不同的子单元公用 ALU 的运算数据通路
      assign {
        mux_op1
       ,mux_op2
      ......
       ,op_cmp_eq
       ,op_cmp_ne
       ,op_cmp_lt
       ,op_cmp_gt
       ,op_cmp_ltu
       ,op_cmp_gtu
       }
    =
```

```
// 来自 Regular-ALU 子单元的运算请求
        ({DPATH_MUX_WIDTH{alu_req_alu}} & {
   ......
        })
// 来自 BJP 子单元的运算请求
      | ({DPATH_MUX_WIDTH{bjp_req_alu}} & {
           bjp_req_alu_op1
          ,bjp_req_alu_op2
          ......
          ,bjp_req_alu_cmp_eq
          ,bjp_req_alu_cmp_ne
          ,bjp_req_alu_cmp_lt
          ,bjp_req_alu_cmp_gt
          ,bjp_req_alu_cmp_ltu
          ,bjp_req_alu_cmp_gtu

        })
// 来自 AGU 模块的运算请求
      | ({DPATH_MUX_WIDTH{agu_req_alu}} & {
         ......
        })
        ;

......

//复用“移位器”

wire ['E203_XLEN-1:0] shifter_in1;
wire [5-1:0] shifter_in2;
wire ['E203_XLEN-1:0] shifter_res;


wire op_shift = op_sra | op_sll | op_srl;

//为了节省面积开销，将右移统一转化为左移操作

assign shifter_in1 = {'E203_XLEN{op_shift}} &
      (
           (op_sra | op_srl) ?
             {
  shifter_op1[00],shifter_op1[01],shifter_op1[02],shifter_op1[03],
  shifter_op1[04],shifter_op1[05],shifter_op1[06],shifter_op1[07],
  shifter_op1[08],shifter_op1[09],shifter_op1[10],shifter_op1[11],
  shifter_op1[12],shifter_op1[13],shifter_op1[14],shifter_op1[15],
  shifter_op1[16],shifter_op1[17],shifter_op1[18],shifter_op1[19],
  shifter_op1[20],shifter_op1[21],shifter_op1[22],shifter_op1[23],
  shifter_op1[24],shifter_op1[25],shifter_op1[26],shifter_op1[27],
  shifter_op1[28],shifter_op1[29],shifter_op1[30],shifter_op1[31]
            } : shifter_op1
      );
assign shifter_in2 = {5{op_shift}} & shifter_op2[4:0];
```

```
    assign shifter_res = (shifter_in1 << shifter_in2);

    wire ['E203_XLEN-1:0] sll_res = shifter_res;
    wire ['E203_XLEN-1:0] srl_res =
                      {
      shifter_res[00],shifter_res[01],shifter_res[02],shifter_res[03],
      shifter_res[04],shifter_res[05],shifter_res[06],shifter_res[07],
      shifter_res[08],shifter_res[09],shifter_res[10],shifter_res[11],
      shifter_res[12],shifter_res[13],shifter_res[14],shifter_res[15],
      shifter_res[16],shifter_res[17],shifter_res[18],shifter_res[19],
      shifter_res[20],shifter_res[21],shifter_res[22],shifter_res[23],
      shifter_res[24],shifter_res[25],shifter_res[26],shifter_res[27],
      shifter_res[28],shifter_res[29],shifter_res[30],shifter_res[31]
                      };


        //复用“异或逻辑门”
    assign xorer_in1 = {'E203_XLEN{xorer_op}} & misc_op1;
    assign xorer_in2 = {'E203_XLEN{xorer_op}} & misc_op2;

    wire ['E203_XLEN-1:0] xorer_res = xorer_in1 ^ xorer_in2;

    wire neq  = (|xorer_res);
    wire cmp_res_ne  = (op_cmp_ne  & neq);
         // It is Equal if it is not Non-Equal
        wire cmp_res_eq  = op_cmp_eq  & (~neq);

      ......


         //复用“加法器”

        // Make sure to use logic-gating to gateoff the
     assign adder_in1 = {'E203_ALU_ADDER_WIDTH{adder_addsub}} & (adder_op1);
    assign adder_in2 = {'E203_ALU_ADDER_WIDTH{adder_addsub}} & (adder_sub ? (
~adder_op2) : adder_op2);
    assign adder_cin = adder_addsub & adder_sub;

    assign adder_res = adder_in1 + adder_in2 + adder_cin;
      ......

          //生成最终的运算数据通路结果，使用and-or的编码方式生成多路选择器
     wire ['E203_XLEN-1:0] alu_dpath_res =
           ({'E203_XLEN{op_or       }} & orer_res )
         | ({'E203_XLEN{op_and      }} & ander_res)
         | ({'E203_XLEN{op_xor      }} & xorer_res)
         | ({'E203_XLEN{op_addsub   }} & adder_res['E203_XLEN-1:0])
         | ({'E203_XLEN{op_srl      }} & srl_res)
         | ({'E203_XLEN{op_sll      }} & sll_res)
         | ({'E203_XLEN{op_sra      }} & sra_res)
         | ({'E203_XLEN{op_mvop2    }} & mvop2_res)
         | ({'E203_XLEN{op_slttu    }} & slttu_res)
```

```
        | ({'E203_XLEN{op_max | op_maxu | op_min | op_minu}} & maxmin_res)
        ;

    //实现 AGU 和 MDV 模块共享的两份 33 位宽的寄存器

  sirv_gnrl_dffl #(33) sbf_0_dffl (sbf_0_ena, sbf_0_nxt, sbf_0_r, clk);
  sirv_gnrl_dffl #(33) sbf_1_dffl (sbf_1_ena, sbf_1_nxt, sbf_1_r, clk);

//寄存器的使能信号选择来自 MDV 还是 AGU 模块
  assign sbf_0_ena =
'ifdef E203_SUPPORT_SHARE_MULDIV //{
      muldiv_req_alu ? muldiv_sbf_0_ena :
'endif//E203_SUPPORT_SHARE_MULDIV}
              agu_sbf_0_ena;
  assign sbf_1_ena =
'ifdef E203_SUPPORT_SHARE_MULDIV //{
      muldiv_req_alu ? muldiv_sbf_1_ena :
'endif//E203_SUPPORT_SHARE_MULDIV}
              agu_sbf_1_ena;

//寄存器的写入数据选择来自 MDV 还是 AGU 模块
  assign sbf_0_nxt =
'ifdef E203_SUPPORT_SHARE_MULDIV //{
      muldiv_req_alu ? muldiv_sbf_0_nxt :
'endif//E203_SUPPORT_SHARE_MULDIV}
              {1'b0,agu_sbf_0_nxt};
  assign sbf_1_nxt =
'ifdef E203_SUPPORT_SHARE_MULDIV //{
      muldiv_req_alu ? muldiv_sbf_1_nxt :
'endif//E203_SUPPORT_SHARE_MULDIV}
              {1'b0,agu_sbf_1_nxt};

//将共享寄存器的值送给 AGU 模块
  assign agu_sbf_0_r = sbf_0_r['E203_XLEN-1:0];
  assign agu_sbf_1_r = sbf_1_r['E203_XLEN-1:0];

//将共享寄存器的值送给 MDV 模块
'ifdef E203_SUPPORT_SHARE_MULDIV //{
  assign muldiv_sbf_0_r = sbf_0_r;
  assign muldiv_sbf_1_r = sbf_1_r;
'endif//E203_SUPPORT_SHARE_MULDIV}
```

8.3.9 高性能乘除法

蜂鸟 E200 系列对于乘除法指令的支持有两种不同的配置。除了第 8.3.8 节中介绍过的小面积多周期乘除法器之外，另外一种配置是高性能的单周期乘法器和独立的除法器。

如果配置了高性能版本的乘除法器，则如图 8-8 所示，乘法器将使用单周期的独立乘法

器位于流水线的第二级；除法器仍然使用多周期除法器，但是不再与 ALU 复用共享的运算数据通路，而是作为长指令具有单独的除法器单元。

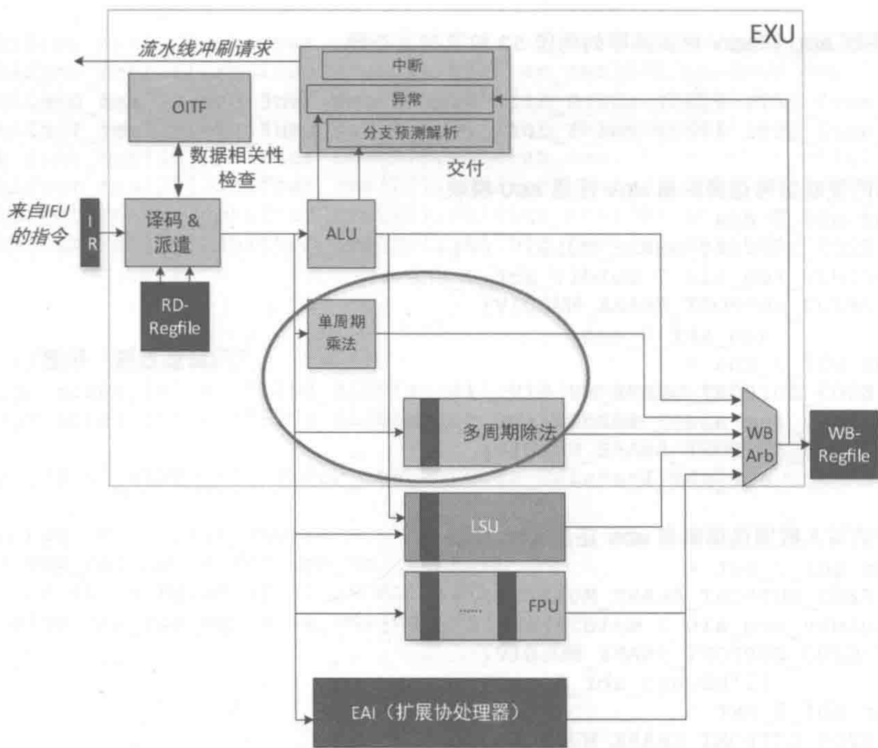


图 8-8 蜂鸟 E200 配置高性能的乘除法单元

由于开源的 E203 处理器核没有配置此高性能版本，因此对于此配置的代码实现，本书在此不做赘述。

8.3.10 浮点单元

蜂鸟 E200 系列也支持 RISC-V 架构的“F”和“D”扩展子集，分别对应单精度浮点和双精度浮点指令子集。

浮点指令由 FPU（Floating Point Unit）运算单元支持。如果配置了 FPU，则如图 8-9 所示，FPU 作为长指令具有单独的运算单元。另外，FPU 还有独立的通用浮点寄存器组。根据 RISC-V 架构定义，如果支持“F”和“D”模块化扩展子集，则需要另外一个独立的浮点寄存器组，包含 32 个通用浮点寄存器。如果仅使用 F 模块的浮点指令子集，则每个通用浮点寄存器的宽度为 32 位；如果使用了 D 模块的浮点指令子集，则每个通用浮点寄存器的宽度为 64 位。

蜂鸟 E200 的 FPU 实现十分高效，具有以下亮点。

- 独立的时钟域和电源域，在不使用之时可以自动关闭各模块时钟，以节省功耗。
- 独立的电源域，在不使用浮点单元之时可以软件控制关闭整个 FPU 的电源，以节省功耗。
- 单精度和双精度浮点指令复用数据通路，在同时支持单精度和双精度浮点指令时，硬件开销仅为双精度浮点运算单元的开销，而不是分别的单精度和双精度两套硬件开销。

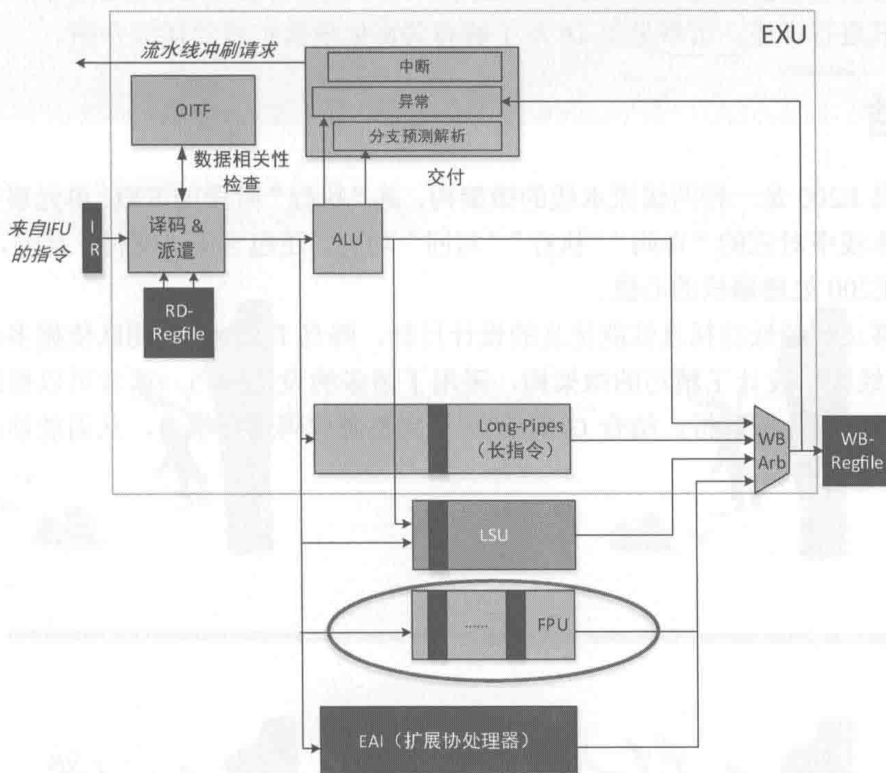


图 8-9　蜂鸟 E200 配置 FPU 支持浮点指令

由于开源的 E203 处理器核没有配置 FPU，因此对于 FPU 的代码实现，本书在此不做赘述。

8.3.11　交付

如第 8.3.2 节所述，指令交付功能也在 EXU 单元完成。由于将交付的功能阐述清楚需要较多篇幅，因此本书单设一章对其进行详述，请参见第 9 章了解有关交付的详细介绍。

8.3.12　写回

如第 8.3.2 节所述，指令结果写回功能也在 EXU 单元完成。由于阐述清楚写回功能需要较多篇幅，因此本书单设一章对其进行详述，请参见第 10 章了解有关写回的详细介绍。

8.3.13 协处理器扩展

可扩展性是 RISC-V 架构最大的亮点之一。如图 8-2 所示，蜂鸟 E200 处理器核的执行阶段也支持协处理器扩展指令。由于协处理器扩展的功能阐述清楚需要较多篇幅，因此本书单设一章对其进行详述，请参见第 16 章了解有关协处理器扩展的详细介绍。

8.3.14 小结

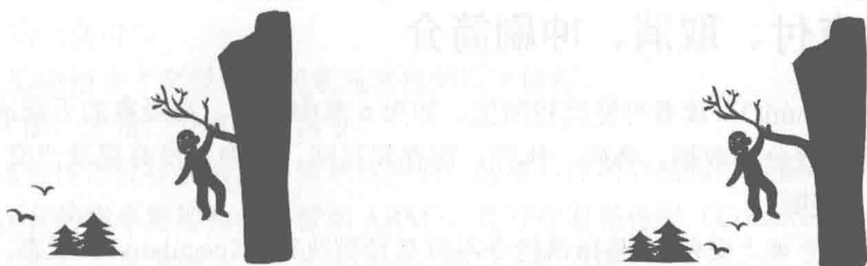
由于蜂鸟 E200 是一种两级流水线的微架构，其“执行”阶段的 EXU 单元事实上包含了经典五级流水线中对应的“译码”“执行”“写回”功能，还包含了“交付”功能，因此 EXU 单元是蜂鸟 E200 处理器核的心脏。

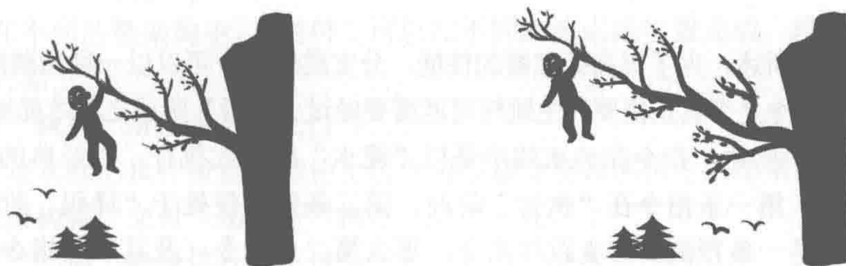
为了能够达到超低功耗且性能优良的设计目标，蜂鸟 E200 研发团队依据多年业界一流 CPU 研发的经验，设计了精巧的微架构，采用了诸多的设计技巧。读者可以借助本章的文字介绍和关键代码片段解析，结合 GitHub 上的完整源代码进行学习，从而能够透彻地理解蜂鸟 E200 的设计精髓。

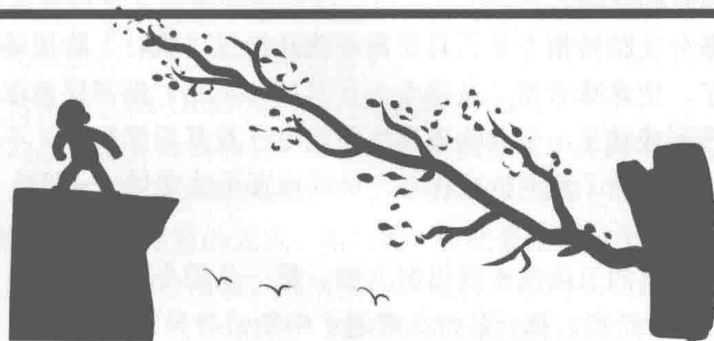


第 9 章　善始者实繁，克终者盖寡——交付

坚持就是胜利







《谏太宗十思疏》中有云：“善始者实繁，克终者盖寡”。意思是认真开始做一件事的人很多，但是能够坚持到最后认真完成的人却寥寥无几。此古语让人联想到指令在流水线中的行为，指令从存储器中读取出来进入流水线开始执行，但却并不是每一条指令都能够被真正的“交付”。

本章将简要介绍处理器交付的功能和常见的策略，并介绍蜂鸟 E200 处理器核交付单元的微架构和源码分析。

9.1 处理器交付、取消、冲刷

9.1.1 处理器交付、取消、冲刷简介

谈及交付（Commit），读者可能比较陌生，如第 6 章中所述，在经典的五级流水线模型中，处理器的流水线分为取指、译码、执行、访存和写回，其中并没有提及“交付”。那么“交付”又有何功能呢？

流水线中的指令被“交付”，是指该指令不再是预测执行（Speculative）状态。它被判定为可以真正地在处理器中被执行，可以对处理器状态产生影响。与“交付”相反的一个名词概念是“取消（Cancel）”，表示该指令最后被判定为需要取消。可能初学的读者还是无法理解，下面列举两种常见的情形阐述“交付”的功能。

（1）情形一

- 如第 7 章中所述，为了提高处理器的性能，分支跳转指令可以以一种预测的形式执行，分支跳转指令是否真正需要产生跳转可能需要经过“执行”阶段之后才能够被确定。
- 如第 6 章中所述，指令在流水线中是以“流水”的形式执行，以经典的五级流水线模型为例，第一条指令在“执行”阶段，第二条指令便处于“译码”阶段。如果第一条指令是一条预测的分支跳转指令，那么第二条指令（及其后续指令）便处于一种预测执行的状态。
- 在第一条分支跳转指令是否真正需要跳转经过“执行”阶段确定之后，如果发现预测错误了，便意味着第二条指令（及其后续指令）都需要被取消掉并放弃执行；如果发现预测成功了，便意味着第二条指令（及其后续指令）不需要被取消掉，可以真正执行，解除了预测执行状态，可以被真正地交付——即被“交付”掉。

（2）情形二

- 同上，以经典的五级流水线模型为例，第一条指令在“执行”阶段，第二条指令便处于“译码”阶段。第一条指令遭遇了中断或者异常（参见第 13 章了解更多中断和

异常的信息)，那么第二条指令和后续的指令便都需要被取消而放弃执行(无法被“交付”，而被“取消”掉)。

谈及处理器中的“交付”，还需介绍另外一个常见的概念——处理器流水线冲刷。如上述的两种情形，当处理器流水线需要将没有“交付”的后续指令全都“取消”掉时，就会造成“流水线冲刷”——就像水流一样将流水线重新冲刷干净，之后重新开始取新的指令。

9.1.2 处理器交付常见实现策略

处理器“交付”的实现非常依赖具体的微架构，常见的实现策略简述如下。

(1) 不管是流水线级数少的简单处理器核，还是流水线级数非常多的高级超标量处理器核，“交付”都通常为顺序判定，理论上只有前一条指令完成了“交付”之后，才会轮到后一条指令的“交付”。

(2) 影响指令“交付”的因素通常包括以下情形。

- 中断，异常，分支预测指令。
  这些情形往往会造成“流水线冲刷，即将后续所有的指令流都取消掉。
- 在有的指令集架构中(譬如 ARM)，还存在着条件码(Conditional Code)。
  因此对于每条指令，只有其条件码满足条件为真才会“交付”，否则会被“取消”(只“取消”它自己，并不会产生造成流水线冲刷而取消后续所有指令)。

(3) 处理器的微架构可以选择一个周期“交付”一条指令(性能较低)或者一个周期“交付”多条指令(性能较高)。

(4) 在不同的微架构中，“交付”可以在不同的流水线位置完成，没有绝对的标准，常见的策略有如下。

- 在“执行”阶段进行“交付”。
  在流水线的执行阶段，理论上已经可以将分支预测指令的结果解析完成，因此有的微架构将“交付”功能在“执行”阶段完成。
- 在“写回”阶段进行“交付”。
  由于有的指令需要多个周期才能写回结果，并可能产生错误异常，因此有的微架构将“交付”功能放在“写回”阶段进行。
- 在重排序交付队列(Re-Order Commit Queue)中进行。
  对于高性能的超标量处理器而言，往往是乱序执行乱序写回(参见第 8.1.5 节了解有关指令发射、派遣、执行和写回顺序的更多信息)，写回往往会使用 ROB(Re-Order Buffer)或者纯物理寄存器的方式，相应地，往往会配备一个较深的重排序交付队列用来缓存乱序执行的指令信息，并对其按序进行“交付”。

蜂鸟 E200 处理器核的“交付”“取消”和“冲刷”实现机制在第 9.3 节予以详细介绍。

9.2 RISC-V 架构特点对于交付的简化

RISC-V 指令集架构具有两个显著的特点，可以大幅简化“交付”的硬件实现。

- 指令没有条件码，因此不需要处理单条指令“取消”的情形。
- 所有的运算指令都不会产生异常。这是 RISC-V 很有意思的特点，大多数的指令集架构都会规定“除法指令除以零”为错误异常，浮点指令也会有若干错误异常。但是 RISC-V 规定这些运算指令一概不产生错误异常，因此在硬件实现上无须担心多周期指令写回结果后会产生错误异常。

综上，在 RISC-V 架构的处理器核中只需要处理如下两类流水线冲刷情形。

- 分支预测指令错误预测造成的后续指令流取消。请参见第 7 章了解更多 RISC-V 分支指令的信息。
- 中断和异常造成的后续指令流取消。请参见第 13 章了解更多 RISC-V 中断和异常的信息。

9.3 蜂鸟 E200 处理器交付硬件实现

基于上述分析，蜂鸟 E200 处理器将“交付”安排在“执行”阶段，如图 9-1 所示。对于每一条指令而言，在蜂鸟 E200 处理器核的流水线中。

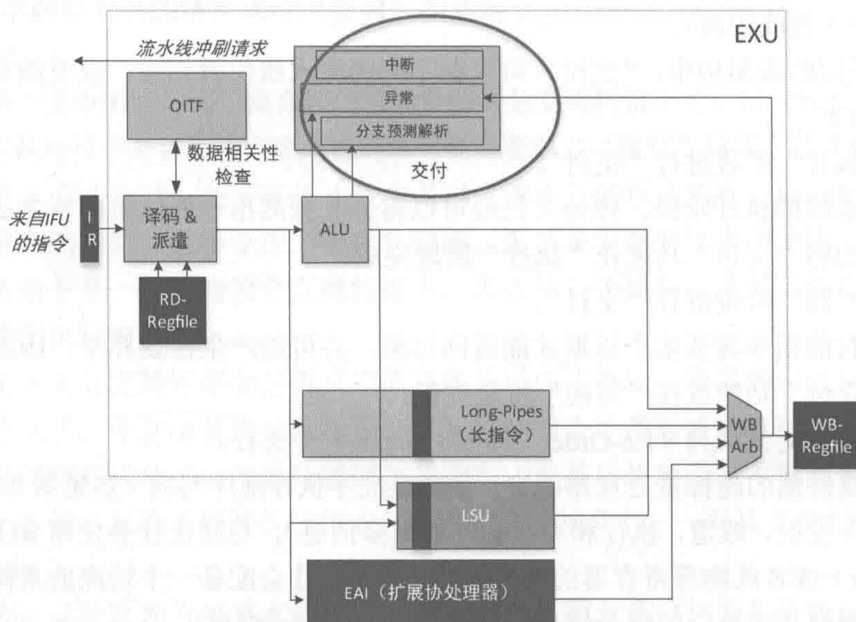


图 9-1　蜂鸟 E200 处理器核的“交付”功能示意图

- 只要前序的指令没有发生“分支预测错误”“中断”或者“异常”，那么就可以判定此条指令能够被成功地“交付”。
- 注意：对于分支预测错误的分支指令自身和遭遇了中断或者异常的指令自身而言，仍然是属于成功“交付”的指令，因为它们自身已经被真正执行且对处理器的状态真正地产生了影响。

9.3.1 分支预测指令的处理

如第7章中所述，在蜂鸟E200处理器核IFU单元中进行预测的分支指令主要为带条件跳转（Conditional Branch）指令类型。

- beq（两个整数操作数等于则跳转）。
- bne（两个整数不相等则跳转）。
- blt（第一个有符号数小于第二个有符号数则跳转）。
- bltu（第一个无符号数小于第二个无符号数则跳转）。
- bge（第一个有符号数大于等于第二个有符号数则跳转）。
- bgeu（第一个无符号数大于等于第二个无符号数则跳转）。

其相关设计要点如下。

- 在蜂鸟E200处理器的IFU单元中对以上带条件跳转指令均采取静态预测，即如果是向后跳转，则预测为跳（Taken）；如果是向前跳转，则预测为不跳（Not Taken）。参见第7.1.4节了解更多静态预测的信息。
- 这些条件跳转指令需要经过“比较”运算才能确定最终是否真的需要跳转，而“比较”运算需由“执行”阶段的ALU完成。相关源代码在e200_opensource目录的结构如下。关于GitHub网站上e200_opensource开源项目的完整代码层次结构详解，请参见第17.1节。

```
e200_opensource
     |----rtl                                // 存放 RTL 的目录
          |----e203                          // E203 核和 SoC 的 RTL 目录
               |----core                     // 存放 e203 Core 的 RTL 代码
                    |----e203_exu_alu_bjp.v    // ALU 的分支跳转指令比较子模块
```

- e203_exu_alu_bjp模块仅处理必要的控制，真正“比较”操作复用的是ALU的数据通路，因此并没有额外的硬件开销。相关源代码片段如下所示。

```
// e203_exu_alu_bjp.v 源代码片段

......
  wire bjp_i_bprdt = bjp_i_info ['E203_DECINFO_BJP_BPRDT ];
```

```
//根据条件跳转指令的类型，向 ALU 运算数据通路发起运算请求
  assign bjp_req_alu_cmp_eq  = bjp_i_info ['E203_DECINFO_BJP_BEQ  ];
  assign bjp_req_alu_cmp_ne  = bjp_i_info ['E203_DECINFO_BJP_BNE  ];
  assign bjp_req_alu_cmp_lt  = bjp_i_info ['E203_DECINFO_BJP_BLT  ];
  assign bjp_req_alu_cmp_gt  = bjp_i_info ['E203_DECINFO_BJP_BGT  ];
  assign bjp_req_alu_cmp_ltu = bjp_i_info ['E203_DECINFO_BJP_BLTU ];
  assign bjp_req_alu_cmp_gtu = bjp_i_info ['E203_DECINFO_BJP_BGTU ];


  assign bjp_o_valid     = bjp_i_valid;
  assign bjp_i_ready     = bjp_o_ready;

//将预测的跳转结果发送给交付模块
  assign bjp_o_cmt_prdt  = bjp_i_bprdt;

//将真实的跳转结果发送给交付模块
        // 如果是无条件跳转（JUMP）指令则一定会跳
        // 如果是条件跳转（Conditional Branch）则会使用 ALU 运算数据通路进行比较运算
        // 的结果
  assign bjp_o_cmt_rslv  = jump ? 1'b1 : bjp_req_alu_cmp_res;
```

```
// e203_exu_alu_dpath.v 源代码片段

......

//不同的模块公用 ALU 的运算数据通路
assign {
     mux_op1
    ,mux_op2
......
    ,op_cmp_eq
    ,op_cmp_ne
    ,op_cmp_lt
    ,op_cmp_gt
    ,op_cmp_ltu
    ,op_cmp_gtu
    }
    =
    // 来自 ALU 的运算请求
       ({DPATH_MUX_WIDTH{alu_req_alu}} & {
    ......
       })
    // 来自 e203_exu_alu_bjp 模块的运算请求
     | ({DPATH_MUX_WIDTH{bjp_req_alu}} & {
            bjp_req_alu_op1
           ,bjp_req_alu_op2
    ......
           ,bjp_req_alu_cmp_eq
           ,bjp_req_alu_cmp_ne
           ,bjp_req_alu_cmp_lt
           ,bjp_req_alu_cmp_gt
```

```
            ,bjp_req_alu_cmp_ltu
            ,bjp_req_alu_cmp_gtu

        })
    // 来自 AGU 模块的运算请求
      | ({DPATH_MUX_WIDTH{agu_req_alu}} & {
          ……
        })
        ;
……

// 进行比较计算

    //复用“异或逻辑门”进行相等比较运算
  assign xorer_in1 = {'E203_XLEN{xorer_op}} & misc_op1;
  assign xorer_in2 = {'E203_XLEN{xorer_op}} & misc_op2;

  wire ['E203_XLEN-1:0] xorer_res = xorer_in1 ^ xorer_in2;

  wire neq  = (|xorer_res);
  wire cmp_res_ne  = (op_cmp_ne  & neq);
      // It is Equal if it is not Non-Equal
  wire cmp_res_eq  = op_cmp_eq  & (~neq);

……


    //复用“加法器”进行比大小运算

     // Make sure to use logic-gating to gateoff the
  assign adder_in1 = {'E203_ALU_ADDER_WIDTH{adder_addsub}} & (adder_op1);
  assign adder_in2 = {'E203_ALU_ADDER_WIDTH{adder_addsub}} & (adder_sub ? (
~adder_op2) : adder_op2);
  assign adder_cin = adder_addsub & adder_sub;

  assign adder_res = adder_in1 + adder_in2 + adder_cin;

      // It is Less-Than if the adder result is negative
  wire cmp_res_lt  = op_cmp_lt  & adder_res['E203_XLEN];
  wire cmp_res_ltu = op_cmp_ltu & adder_res['E203_XLEN];
      // It is Greater-Than if the adder result is postive
  wire op1_gt_op2  = (~adder_res['E203_XLEN]);
  wire cmp_res_gt  = op_cmp_gt  & op1_gt_op2;
  wire cmp_res_gtu = op_cmp_gtu & op1_gt_op2;

  assign cmp_res = cmp_res_eq
                 | cmp_res_ne
                 | cmp_res_lt
                 | cmp_res_gt
                 | cmp_res_ltu
                 | cmp_res_gtu;
```

```
//将比较器的计算结果返回给 e203_exu_alu_bjp 模块
 assign bjp_req_alu_cmp_res = cmp_res;
```

- ALU 在计算出是否需要跳转的结果之后，发送给“交付”模块。“交付”模块则根据预测的结果和真实的结果进行判断。如果预测和真实的结果相符，则意味着预测成功，不会进行流水线冲刷；反之，则预测失败需要进行流水线冲刷。相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                // 存放 RTL 的目录
        |----e203                           // E203 核和 SoC 的 RTL 目录
            |----core                       // 存放 e203 Core 的 RTL 代码
                |----e200_exu_commit.v            // 交付模块
                |----e200_exu_branchslv.v       // 分支预测指令交付模块
```

- e203_exu_branchslv 是 e203_exu_commit 模块的子模块，用于对分支预测指令的真实结果进行判断。相关源代码片段如下所示。

```
// e203_exu_branchslv.v 源代码片段

……

  wire brchmis_need_flush = (
// 如果预测的结果和真实的结果不相符，则需要产生流水线冲刷
        (cmt_i_bjp & (cmt_i_bjp_prdt ^ cmt_i_bjp_rslv))
……
     );

  assign brchmis_flush_req_pre = cmt_i_valid & brchmis_need_flush;

  assign brchmis_flush_pc =
  //如果是预测了需要跳转，但是实际结果显示不需要跳转，则流水线冲刷重新取指的新 PC 指向此跳转指令的下一个指令 。通过将此跳转指令的 PC 加上 4（如果此指令是 32 位指令）或者 2（如果此指令是 16 位）计算其下一个指令的 PC。
                              (cmt_i_fencei | (cmt_i_bjp & cmt_i_bjp_prdt)) ?
  (cmt_i_pc + (cmt_i_rv32 ? 'E203_PC_SIZE'd4 : 'E203_PC_SIZE'd2)) :
  //如果是预测了不需要跳转，但是实际结果显示需要跳转，则流水线冲刷重新取指的新 PC 指向此跳转指令跳转目标地址。通过将此跳转指令的 PC 加上跳转目标偏移量计算目标地址。
                              (cmt_i_bjp & (~cmt_i_bjp_prdt)) ?
  (cmt_i_pc + cmt_i_imm['E203_PC_SIZE-1:0]) :
……
```

9.3.2 中断和异常的处理

有关蜂鸟 E200 处理器对于中断和异常的详细实现，请参见第 13 章。

9.3.3 多周期执行指令的交付

对于普通的单周期指令而言，其“交付”和“写回”操作在“执行”阶段的同一个周期内完成。而对于执行周期超过一个周期的多周期指令而言，其“交付”同样在“执行”阶段完成，但是“写回”操作则需在后续的周期内写回。请参见第 10 章了解更多蜂鸟 E200 处理器核的“写回”实现细节。

要注意以下两种情况。

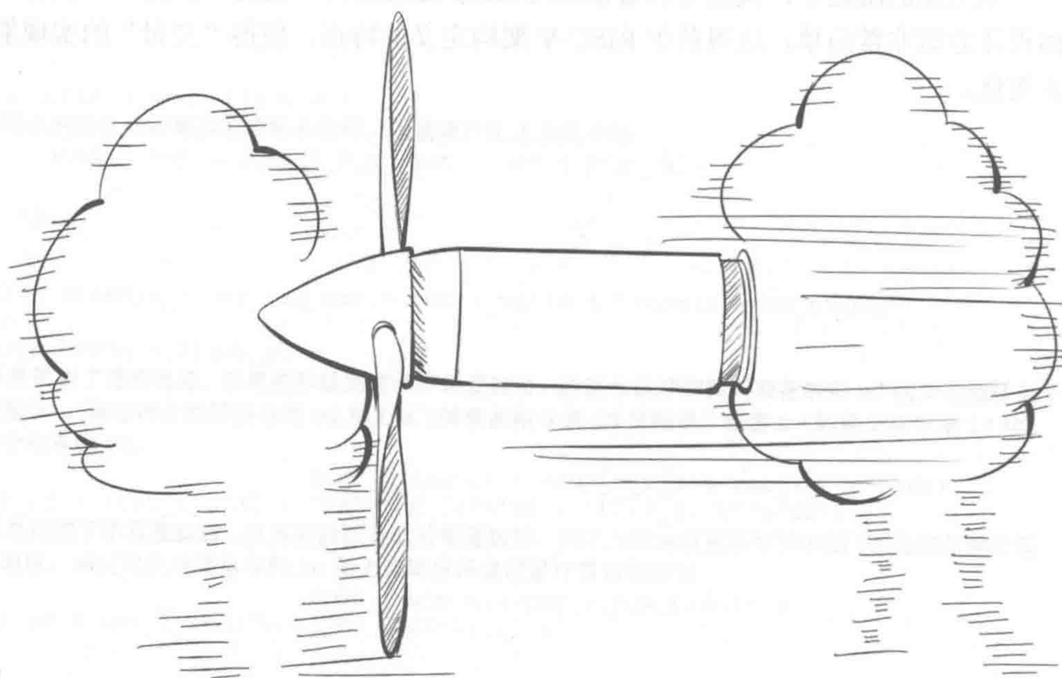
- 虽然有的长指令（譬如 EAI 协处理器指令和 Load、Store 指令）也会在写回时产生错误异常，但是按照 RISC-V 架构规定这两种异常可以当作异步异常处理，有关蜂鸟 E200 处理器对于中断和异常的详细实现，请参见第 13 章。
- 而所有其他的常规多周期指令（譬如除法和浮点指令），RISC-V 架构规定其不会产生任何异常。

9.3.4 小结

从上面的阐述中，我们可以看出蜂鸟 E200 处理器将“交付”安排在“执行”阶段，整体设计方案非常简单，这得益于 RISC-V 架构定义的特点，使得“交付”的实现能够得到极大简化。

第10章　让子弹飞一会儿——写回

让子弹飞一会儿



如第 8 章中所述，指令在蜂鸟 E200 流水线中的派遣点被派遣到不同运算单元执行，就像从枪膛中射出的子弹一样。让子弹飞一会儿，直到它落地的那一刻，从不同的运算单元执行完毕后的指令也最终都会将其计算结果写回（Write-Back）通用寄存器组（Regfile）。

本章将简要介绍处理器的“写回”功能和常见的策略，并介绍蜂鸟 E200 处理器核的“写回”硬件实现微架构和源码分析。

10.1 处理器的写回

10.1.1 处理器写回功能简介

如第 6 章中所述，在经典的五级流水线模型中，处理器的流水线分为取指、译码、执行、访存以及写回。写回是流水线的最后一级，主要的作用是将指令的运算结果写回到通用寄存器组。

10.1.2 处理器写回常见策略

在第 8 章中对指令发射、派遣、执行、写回的顺序和常见策略已予以详述，本节在此不做赘述。

10.2 蜂鸟 E200 处理器的写回硬件实现

蜂鸟 E200 处理器的写回策略是一种因地制宜的混合策略。蜂鸟 E200 处理器核的写回硬件实现不仅保持面积最小化的原则，还能取得不错的性能，其核心思想如下。

- 将指令划分为单周期指令和长指令两大类，参见第 8.3.7 节了解有关单周期指令和长指令的分类信息。
- 将长指令的“交付”和“写回”分开，使得即便执行了多周期长指令，仍然不会阻塞流水线，让后续的单周期指令仍然能够顺利地写回和交付。

涉及的模块如图 10-1 中的圆圈区域所示，主要包含如下几个组件：最终写回仲裁（Final Write-Back Arbitration）、长指令写回仲裁（Long-Pipes Instructions Write-Back Arbitration）和 OITF（Outstanding Instruction Track FIFO）。下文将分别予以介绍。

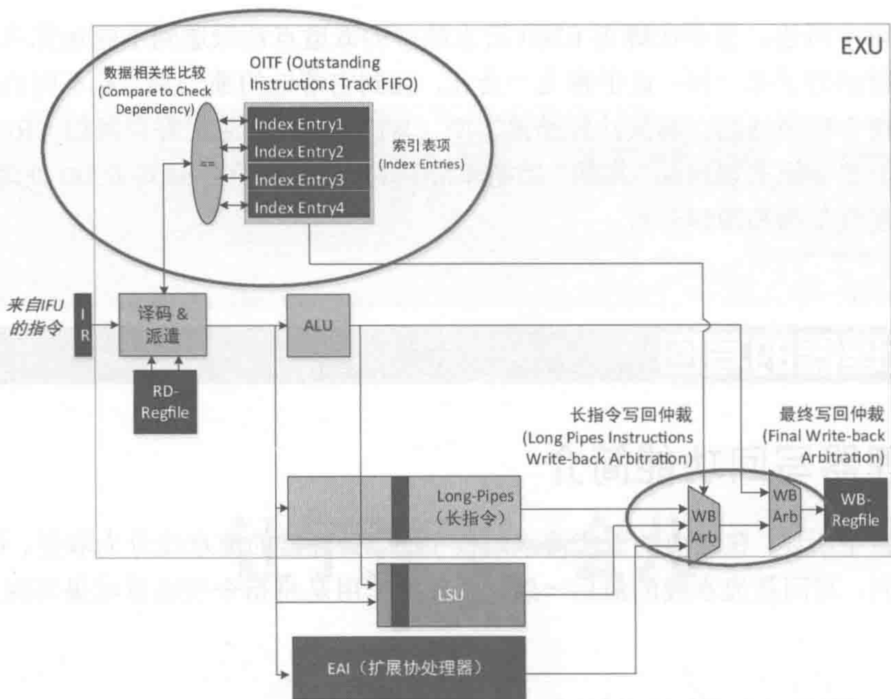


图 10-1 蜂鸟 E200 处理器核的 Write-Back 功能示意图

10.2.1 最终写回仲裁

蜂鸟 E200 处理器有两级写回仲裁（WB-Arb）模块，其中之一是最终写回仲裁模块，其要点如下。

- 如图 10-1 所示，“最终写回仲裁”主要用于仲裁所有单周期指令的写回（来自于 ALU 模块）和所有长指令的写回（来自于“长指令写回仲裁”模块），仲裁采用优先级仲裁的方式。由于长指令的执行周期比较长，因此其明显比正在写回的 ALU 指令在程序流中处于更早的位置，所以长指令的写回比单周期指令的写回具有更高的优先级。
- 如果在没有长指令写回的空闲周期，来自 ALU 的单周期指令则可以随便写回。这也就意味着，在程序流中处于更迟位置的单周期指令可以比更早位置的长指令先写回 Regfile（如果没有数据相关性），所以说蜂鸟 E200 处理器也具备乱序写回的能力。
- “最终写回仲裁”模块相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                              // 存放 RTL 的目录
        |----e203                         // E203 核和 SoC 的 RTL 目录
            |----core                     // 存放 e203 Core 的 RTL 代码
                |----e203_exu_wbck.v      // 最终写回仲裁模块
```

- “最终写回仲裁”模块的源代码片段如下所示。

```
// e203_exu_wbck.v 源代码片段
......
  // 所有单周期指令的写回（来自于 ALU 模块）
  //////////////////////////////////////////////////////////////
  // The ALU Write-Back Interface
  input  alu_wbck_i_valid,                          //写回握手请求信号
  output alu_wbck_i_ready,                          //写回握手反馈信号
  input  ['E203_XLEN-1:0] alu_wbck_i_wdat,          //写回的数据值
  input  ['E203_RFIDX_WIDTH-1:0] alu_wbck_i_rdidx,  //写回的寄存器索引值

  // 所有长指令的写回（来自于长指令写回仲裁模块）
  //////////////////////////////////////////////////////////////
  // The Longp Write-Back Interface
  input  longp_wbck_i_valid,                              //写回握手请求信号
  output longp_wbck_i_ready,                              //写回握手反馈信号
  input  ['E203_FLEN-1:0] longp_wbck_i_wdat,              //写回的数据值
  input  ['E203_RFIDX_WIDTH-1:0] longp_wbck_i_rdidx,      //写回的寄存器索引值


  // 仲裁后写回 Regfile 的接口。
  //////////////////////////////////////////////////////////////
  // The Final arbitrated Write-Back Interface to Regfile
  output  rf_wbck_o_ena,                               //写使能
  output  ['E203_XLEN-1:0] rf_wbck_o_wdat,             //写回的数据值
  output  ['E203_RFIDX_WIDTH-1:0] rf_wbck_o_rdidx,     //写回的寄存器索引值
......

  // 使用优先级仲裁，如果两种指令同时写回，长指令具有更高的优先级。
  // The ALU instruction can write-back only when there is no any
  //  long pipeline instruction writing-back
  //    * Since ALU is the 1 cycle instructions, it have lowest
  //      priority in arbitration
  wire wbck_ready4alu = (~longp_wbck_i_valid);
  wire wbck_sel_alu = alu_wbck_i_valid & wbck_ready4alu;
 // The Long-pipe instruction can always write-back since it have high priority
 wire wbck_ready4longp = 1'b1;
 wire wbck_sel_longp = longp_wbck_i_valid & wbck_ready4longp;
......

 assign alu_wbck_i_ready   = wbck_ready4alu   & wbck_i_ready;
 assign longp_wbck_i_ready = wbck_ready4longp & wbck_i_ready;

......
```

10.2.2 OITF 和长指令写回仲裁

OITF 和长指令写回仲裁模块协同合作完成所有长指令的写回操作，其要点如下。

- 如图 10-1 所示，“长指令写回仲裁”主要用于仲裁不同的长指令之间的写回，譬如来自于 LSU、乘除法器、FPU 和 EAI 协处理器等的写回。由于这些不同的长指令执行的周期数各不相同，甚至有的执行周期数是动态的，因此无法轻易判断这些指令的先后关系，所以需要记录这些指令的先后关系。
- 如图 10-1 所示，OITF（Outstanding Instruction Track FIFO）即用于记录这些长指令的信息。

  OITF 本质上是一个先入先出的 FIFO，在流水线的派遣点，每次派遣一个长指令，则会在 OITF 中分配一个表项（Entry），这个表项的 FIFO 指针（Pointer）便作为这个长指令的 ITAG（Instruction Tag）。

  派遣的长指令不管被派遣到任何运算单元，都会携带者这个 ITAG，长指令的运算单元在完成了运算后将结果写回之时，也要携带其对应的 ITAG。
- OITF 的深度便决定了能够派遣的滞外（Outstanding）长指令的个数。这些长指令写回时，理论上其实无须严格地按照其派遣顺序，只需要在有寄存器冲突的情形时才严格遵循顺序即可，其他情形时可以乱序写回。但是为了硬件实现的简洁，蜂鸟 E200 选择了简单的严格参照 OITF 的顺序写回。

  由于 OITF 是一个先入先出的 FIFO，因此 FIFO 的读指针（Read Pointer）会指向最先进入此 FIFO 的表项，通过使用此“读指针”作为长指令写回仲裁的选择参考，就可以保证不同长指令的写回顺序和派遣顺序严格一致。

  每次从“长指令写回仲裁”模块成功地写回一个长指令之后，便将此指令在 OITF 中的表项去除，即从 FIFO 退出（先入先出）完成其历史使命。
- 由于有的长指令可能发生执行错误，因此需要产生异常。所以“长指令写回仲裁”模块需要和“交付”模块进行接口触发异常。如果长指令产生了异常，则不会真的写回 Regfile。有关蜂鸟 E200 的异常处理实现请参见第 13 章。
- OITF 和“长指令写回仲裁”模块相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                              // 存放 RTL 的目录
        |----e203                         // E203 核和 SoC 的 RTL 目录
            |----core                     // 存放 e203 Core 的 RTL 代码
                |----e200_exu_disp.v          // 指令派遣模块
```

```
                    |----e200_exu_oitf.v       // OITF 模块
                    |----e203_exu_longpwbck.v  // Long-Pipes Instructions
                                               // Write-Back Arbitration 模块
```

- OITF 模块的源代码分析请参见第 8.3.7 节。派遣模块和“长指令写回仲裁”模块的相关源代码片段如下所示。

```
// e203_exu_disp.v 源代码片段

  ......

  //在派遣点产生 OITF 分配表项的使能信号
  // 如果当前派遣的指令为一个长指令则产生此使能信号
  assign disp_oitf_ena = disp_o_alu_valid & disp_o_alu_ready & disp_alu_lon
gp_real;
  ......
```

```
// e203_exu_longpwbck.v 源代码片段

  //来自整数除法单元的写回接口
  'ifdef E203_SUPPORT_INDEP_MULDIV //{
    //////////////////////////////////////////////////////////////
    // The DIV Write Interface
    input  div_wbck_i_valid,                          //写回握手请求信号
    output div_wbck_i_ready,                          //写回握手反馈信号
    input  ['E203_XLEN-1:0] div_wbck_i_wdat,          //写回的数据值
    input  div_wbck_i_err,                            //写回的异常错误指示
    input  ['E203_ITAG_WIDTH-1:0] div_wbck_i_itag,    //写回指令的 ITAG
  'endif//}

  //来自 LSU 单元的写回接口
    //////////////////////////////////////////////////////////////
    // The LSU Write-Back Interface
    input   lsu_wbck_i_valid,                             //写回握手请求信号
    output  lsu_wbck_i_ready,                             //写回握手反馈信号
    input   ['E203_XLEN-1:0] lsu_wbck_i_wdat,             //写回的数据值
    input   ['E203_ITAG_WIDTH -1:0] lsu_wbck_i_itag,      //写回指令的 ITAG
    input   lsu_wbck_i_err ,                              //写回的异常错误指示
    input   lsu_cmt_i_buserr ,                            //访存错误异常错误指示
    input   ['E203_ADDR_SIZE -1:0] lsu_cmt_i_badaddr,     //产生访存错误的地址
    input   lsu_cmt_i_ld,                                 //产生访存错误为 Load 指令
    input   lsu_cmt_i_st,                                 //产生访存错误为 Store 指令

  // 仲裁后的写回接口，通给最终写回仲裁模块。
    //////////////////////////////////////////////////////////////
    // The Long pipe instruction Wback interface to final wbck module
    output longp_wbck_o_valid, // Handshake valid
```

```
    input  longp_wbck_o_ready, // Handshake ready
    output ['E203_FLEN-1:0] longp_wbck_o_wdat,
    output ['E203_RFIDX_WIDTH -1:0] longp_wbck_o_rdidx,

  // 仲裁后的异常接口，通给交付模块。
    // The Long pipe instruction Exception interface to commit stage
    output  longp_excp_o_valid,
    input   longp_excp_o_ready,
    output  longp_excp_o_insterr,
    output  longp_excp_o_ld,
    output  longp_excp_o_st,
    output  longp_excp_o_buserr ,
    output ['E203_ADDR_SIZE-1:0] longp_excp_o_badaddr,
  ……
    // 使用 OITF 的读指针（信号 oitf_ret_ptr）作为
    // 长指令写回仲裁的选择参考。

      // The Long-pipe instruction can write-back only when it's itag
      //   is same as the itag of toppest entry of OITF
    wire wbck_ready4lsu = (lsu_wbck_i_itag == oitf_ret_ptr) & (~oitf_empty);
    wire wbck_sel_lsu = lsu_wbck_i_valid & wbck_ready4lsu;
  'ifdef E203_SUPPORT_INDEP_MULDIV //{
    wire wbck_ready4div = (div_wbck_i_itag == oitf_ret_ptr) & (~oitf_empty);
    wire wbck_sel_div = div_wbck_i_valid & wbck_ready4div;
  'endif//}

  ……

    // 只有没有异常错误的指令才需要写回 Regfile

     // If the instruction have no error and it have the rdwen, then it need to
     //   write back into regfile, otherwise, it does not need to write regfile
    wire need_wbck = wbck_i_rdwen & (~wbck_i_err);

    // 产生了异常错误的指令需要和交付模块接口

      // If the long pipe instruction have error result, then it need to handshake
      //   with the commit module.
    wire need_excp = wbck_i_err;

    // 需要保证交付模块和最终写回仲裁模块同时能够接受
    assign wbck_i_ready =
         (need_wbck ? longp_wbck_o_ready : 1'b1)
       & (need_excp ? longp_excp_o_ready : 1'b1);

    //通给最终写回仲裁模块的握手请求
    assign longp_wbck_o_valid = need_wbck & wbck_i_valid & (need_excp ? longp
_excp_o_ready : 1'b1);

    //通给交付模块的握手请求
    assign longp_excp_o_valid = need_excp & wbck_i_valid & (need_wbck ? longp
```

```
_wbck_o_ready : 1'b1);


    // 每次从长指令写回仲裁模块成功的
    // 写回一个长指令之后，便将此指令在 OITF 中的表项去除，即从 FIFO 退出（先入先出）完成其
    //  历史使命。
    // 以下信号即为成功写回一个长指令的使能信号。
    assign oitf_ret_ena = wbck_i_valid & wbck_i_ready;
```

10.2.3 小结

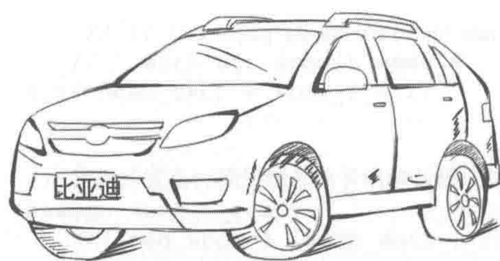
从上述的阐述中，我们可以看出蜂鸟 E200 处理器的写回是一种混合策略，体现在如下方面。

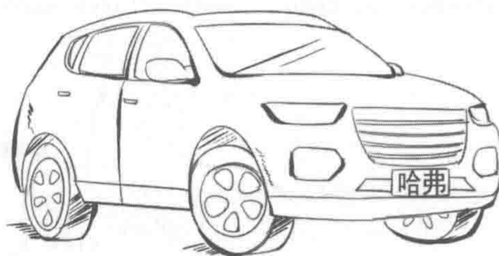
- 如果仅讨论单周期指令，蜂鸟 E200 的策略属于“顺序发射，顺序执行，顺序写回”。
- 如果仅讨论长指令（由不同的长指令运算单元执行），蜂鸟 E200 的策略属于“顺序发射，乱序执行，顺序写回”。
- 如果将单周期指令和长指令统一考虑，蜂鸟 E200 的策略则属于“顺序发射，乱序执行，乱序写回”。

所谓“法无长势，运用之妙，存乎一心”，体系结构设计本来就是一种非常灵活和创新的学问，因此无须太过拘泥于固定的分类。蜂鸟 E200 处理器作为超低功耗处理器核，按道理可以选择最简单的顺序执行、顺序写回的策略，但是我们却选择追求极小面积的同时，也兼顾性能。通过将指令划分为单周期指令和长指令两大类，并且将“交付”和“写回”分开的方式，使得即便执行了多周期长指令，仍然不会阻塞流水线。后续的大多数单周期指令仍然能够顺利地“写回”和“交付”，并且通过 OITF 记录的顺序将长指令的写回单独处理。硬件实现既简洁，而又不失性能，这便是“运用之妙，存乎一心”的体现。

第 11 章 哈弗还是比亚迪——存储器架构

哈弗还是比亚迪？看上去都挺好！





本章的标题虽然是“哈弗还是比亚迪”，但是本书和汽车毫无关系，之所以如此标题，是因为本章将讨论选择“哈佛”还是“冯•诺依曼”结构。“哈弗”只是“哈佛”结构的一个谐音，此梗可能有点冷。

谈到哈佛结构，熟悉计算机体系结构的读者可能已经联想到本章的主题，本章将简要介绍处理器的存储器（Memory）架构，并介绍蜂鸟 E200 处理器核存储器子系统的微架构和源码分析。

11.1 存储器架构概述

在介绍蜂鸟 E200 处理器核的存储器子系统之前，本节先讨论有关存储器架构的几个常见话题。

11.1.1 谁说处理器一定要有缓存

谈及处理器的存储器子系统，讨论得最多的莫过于缓存（Cache）。缓存几乎可以认为是处理器微架构中最复杂的部分，常见的缓存基础知识如下。

- 缓存的映射类型。常见类型如直接映射、全相联映射、组相联映射等。
- 缓存的写回策略。常见类型如写穿通、写分配等。
- 缓存使用物理地址还是虚拟地址索引。常见类型如 PIPT、VIPT 等。
- 缓存的标签（Tag）和数据（Data）的组织顺序。常见的如串行组织、并行组织等。
- 多级缓存的组织和管理。
- 在多核架构下缓存一致性（Cache Coherency）。

有关缓存的基础知识和设计技巧，本书限于篇幅不做赘述，感兴趣的读者可以参见维基百科上关于缓存的词条网页（请在维基百科搜索“CPU cache”）。

缓存虽然很火，但是谁说处理器一定要有缓存？也许很多读者一直认为缓存是处理器中必不可少的部分，但是众多极低功耗的处理器其实并没有配备缓存，主要是出于如下几个方面的原因。

（1）无法保证实时性。

- 这是缓存不被使用的最主要原因。缓存是一种缓存机制，利用软件程序的时间局部性和空间局部性，将空间巨大的存储器数据动态映射到容量有限的缓存中，可以将访问存储器的平均延迟降低到最小。
- 由于缓存的容量是有限的，因此访问缓存存在着相当大的不确定性。一旦缓存不命中（Cache-Miss），则需要从外部的存储器中存取数据，造成较长的延迟。在对实时性要求高的场景中，处理器的反应速度必须有最可靠的实时性。如果使用了缓存，则无法保证这一点。

- 大多数极低功耗处理器应用的场景都应用于实时性较高的场景，因此更加倾向于使用延迟确定的 ITCM（Instruction Tightly Coupled Memory）或者 DTCM（Data Tightly Coupled Memory），有关 ITCM 和 DTCM 的相关信息在第 11.1.3 节中专门介绍。

（2）软件规模较小。

- 大多数极低功耗处理器均应用于深嵌入式领域。此领域中的软件代码规模一般较小，所需要的数据段也较小，使用几十 KB 的片上 SRAM 或者 ITCM/DTCM 便可以满足其需求，因此缓存能够缓存空间巨大存储器数据的优点在此变得无用武之地了。

（3）面积功耗大。

- 缓存的设计难度相比 ITCM 和 DTCM 要大很多，消耗的面积资源和带来的功耗损失也更大。而极低功耗处理器更加追求小面积和能效比，因此更倾向于使用 ITCM 和 DTCM。

基于如上几个原因，目前主流的极低功耗处理器其实都没有使用缓存。以 ARM Cortex-M 为例，如表 11-1 所示，仅有最高端的 Cortex-M7 配备了缓存，须知 Cortex-M7 采用了六级流水线双发射的处理器核，已经不算是极低功耗处理器了。

表 11-1　ARM Cortex-M 的缓存配置情况

Cortex-M 系列型号	是否有缓存
ARM Cortex-M0+	无
ARM Cortex-M0	
ARM Cortex-M3	
ARM Cortex-M4	
ARM Cortex-M7	有

11.1.2 处理器一定要有存储器

虽然处理器不一定需要缓存，但是处理器是一定需要存储器的。

根据计算机奠基人之一的科学家冯•诺依曼（John Von•Neumann）提出的计算机体系结构模型，如图 11-1 所示，计算机必须具备五大组成部分：输入设备、输出设备、存储器、运算器和控制器。其中，运算器和控制器可以归于处理器核的范畴，其运行的指令和所需要的数据都必须来自于存储器。

熟悉计算机体系结构的读者一定知道处理器访问存储器的策略，在理论上存在着冯•诺依曼结构和哈佛（Harvard）结构两种。

（1）冯•诺依曼结构也称普林斯顿结构，是一种将指令存储器和数据存储器合并在一起的存储器结构。程序的指令存储地址和数据存储地址指向同一个存储器的不同物理位置。

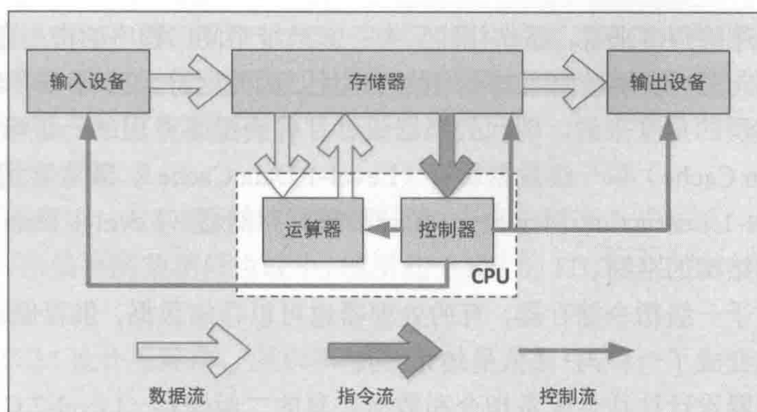


图 11-1 冯•诺依曼定义的计算机体系模型

（2）哈佛结构是一种将指令存储器和数据存储器分开的存储器结构。如图 11-2 所示，它的主要特点如下。

- 将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问。
- 与两个存储器相对应的是两条独立的指令总线和数据总线。这种分离的总线使得处理器可以在一个周期内同时获得指令字（来自指令存储器）和操作数（来自数据存储器），从而提高了执行速度和数据的吞吐率。
- 由于指令和数据存储在两个分开的物理空间中，因此取址和执行能完全并行。

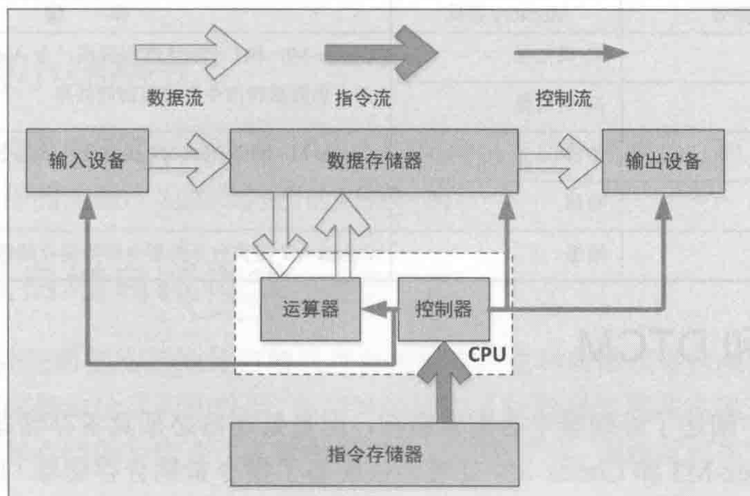


图 11-2 哈佛结构

而在实际的现代处理器设计中，冯•诺依曼结构和哈佛结构的概念界限已经变得越来越微妙，而且模糊，具体阐述如下。

- 从软件程序的角度来看，系统往往只有一套地址空间，程序的指令存储地址和数据存储地址指向同一套地址空间的不同物理地址，因此这符合冯•诺依曼体系结构的准则。
- 从硬件实现的角度来看，现代处理器设计往往会配备专用的一级指令缓存（Level-1 Instruction Cache）和一级数据缓存（Level-1 Data Cache），或者专用的一级指令存储器（Level-1 Instruction Memory）和一级数据存储器（Level-1 Data Memory），因此符合哈佛结构的准则。
- 即便是对于一级指令储存器，有的处理器也可以存储数据，供存储器读写指令访问。因此这也变成了一种冯•诺依曼体系结构。
- 现代处理器设计往往会配备指令和数据共享的二级缓存（Level-2 Cache），或者共享的二级存储器（Level-2 Memory）。在二级存储器中，程序的指令存储地址和数据存储地址指向同一套地址空间的不同物理地址，并且共享读写访问通道，因此符合冯•诺依曼结构的准则。

综上原因，可以看出冯•诺依曼结构和哈佛结构并不是一种非此即彼的选择，也无须按照这两种架构严格区分和划分现代的处理器。在处理器的实际设计中，只需要明白其优缺点，灵活地加以运用即可。

对 ARM Cortex-M 系列处理器核而言，按照其存储器访问接口的数目作为划分冯•诺依曼结构或哈佛结构的标准，总结如表 11-2 所示。

表 11-2 ARM Cortex-M 的存储器架构类型

Cortex-M 系列型号	Memory 架构	详　情
ARM Cortex-M0+	冯•诺依曼	Cortex-M0+和 Cortex-M0 仅提供一个 AHB-Lite 存储器访问接口，供数据和指令存储器访问共享
ARM Cortex-M0	冯•诺依曼	
ARM Cortex-M3	哈佛	Cortex-M3 和 Cortex -M4 提供分开的指令和数据存储器访问接口
ARM Cortex-M4	哈佛	
ARM Cortex-M7	哈佛	Cortex-M7 提供分开的指令和数据存储器访问接口

11.1.3 ITCM 和 DTCM

在上述小节中阐述了处理器未必需要缓存，但是处理器必须具备存储器（Memory）。作为典型代表，Cortex-M3 和 Cortex-M4 处理器核配备了指令紧耦合存储器（Instruction Tightly Coupled Memory，ITCM）和数据紧耦合存储器（Data Tightly Coupled Memory，DTCM），相比于缓存（Cache）而言更加适合嵌入式低功耗处理器，原因如下。

（1）能够保证实时性。

ITCM 和 DTCM 被映射到不同的地址区间，处理器的访问使用明确的地址映射的方式访

问 ITCM 和 DTCM。由于 ITCM 和 DTCM 并不是缓存机制，不存在着缓存不命中的情况，其访问的延迟是明确可知的，因此程序的执行过程能够得到明确的性能结果。在实时性要求高的场景，处理器的反应速度能够取得可靠的实时性。

（2）能够满足软件需求。

如第 11.1.1 节所述，大多数极低功耗处理器均应用于深嵌入式领域，此领域中的软件代码规模一般较小，所需要的数据段也较小，使用几十 KB 的 ITCM/DTCM 便可以满足其需求。

（3）面积功耗小。

ITCM 和 DTCM 设计很简单，面积和功耗更小。

11.2 RISC-V 架构特点对于存储器访问指令的简化

上一节讨论了处理器存储器的相关背景和技术，存储器是每个处理器必不可少的一环。

在第 2 章中曾经总结性地探讨过 RISC-V 架构追求简化硬件的哲学，具体对于存储器访问指令而言，RISC-V 架构的如下特点可以大幅简化硬件实现。

- 仅支持小端格式。
- 无地址自增自减模式。
- 无“一次读多个数据（Load-Multiple）”和“一次写多个数据（Store-Multiple）”指令。

下文分别予以论述。

11.2.1 仅支持小端格式

因为现在的主流应用是小端格式，RISC-V 架构仅支持小端模式，而不支持大端格式，因此可以简化硬件的实现，无须做特别的数据转换。

11.2.2 无地址自增自减模式

很多 RISC 处理器都支持地址自增或者自减模式，这种自增或者自减的模式能够提高处理器访问连续存储器地址区间的性能，但同时也增加了处理器的硬件实现难度。由于 RISC-V 架构的存储器读和存储器写指令不支持地址自增自减的模式，因此可以很大程度上简化地址的生成逻辑。

11.2.3 无“一次读多个数据”和“一次写多个数据”指令

很多 RISC 架构定义了一次写多个寄存器到存储器中，或者一次从存储器中读多个寄存

器的指令，这样的好处是一条指令就可以完成很多事情。但是这种“一次读多个数据”和“一次写多个数据”指令的弊端是会让处理器的硬件设计非常复杂，增加硬件的开销，也可能损伤时序，无法提高处理器的主频。而 RISC-V 没有定义此类指令，使得硬件设计非常简单。

11.3 RISC-V 架构的存储器相关指令

本节将介绍 RISC-V 架构的存储器访问相关指令。

11.3.1 Load 和 Store 指令

与所有的 RISC 处理器架构一样，RISC-V 架构使用专用的存储器读（Load）指令和存储器写（Store）指令访问存储器，其他的普通指令无法访问存储器。

RISC-V 架构定义了 7 条存储器读指令和存储器写指令，分别为 LH、LHU、LB、LBU、LW、SB、SH、SW，用于支持以一个字节、半字、单字为单位的存储器读写操作。RISC-V 架构推荐使用地址对齐的存储器读写操作，但是也支持地址非对齐的存储器操作 RISC-V 架构，处理器可以选择用硬件来支持，也可以选择用软件异常服务程序来支持。

有关 Load 和 Store 指令的详细定义，请参见附录 A14.2 节。

11.3.2 Fence 指令

RISC-V 架构采用松散存储器模型（Relaxed Memory Model），松散存储器模型对于访问不同地址的存储器读写指令的执行顺序不作要求，除非使用明确的存储器屏障指令。有关存储器模型和存储器屏障指令的更多信息，请参见附录 A13。

RISC-V 架构定义了 Fence 和 Fence.I 两条存储器屏障指令，用于强行界定存储器访问的顺序，其定义如下。

- 在程序中，如果添加了一条 Fence 指令，则 Fence 指令能够保证“在 Fence 之前所有指令造成的访存结果”必须比“在 Fence 之后所有指令造成的访存结果”先被观测到。
- 在程序中，如果添加了一条 Fence.I 指令，则“在 Fence.I 之后所有指令的取指令操作”一定能够观测到“在 Fence.I 之前所有指令造成的访存结果”。

有关 Fence 和 Fence.I 指令的详细定义，请参见附录 A14.2 节。

11.3.3 “A”扩展指令

RISC-V 架构定义了一种扩展指令子集（由 A 字母表示），主要用于支持在多线程情形

下访问存储器的原子（Atomic）操作或者同步操作，其包括两类指令。

- Atomic Memory Operation (AMO) 指令。
- Load-Reserved 和 Store-Conditional 指令。

有关以上两类指令的详细定义，请参见附录 A14.5 节。

11.4 蜂鸟 E200 处理器存储器子系统硬件实现

11.4.1 存储器子系统总体设计思路

蜂鸟 E200 处理器核的存储器子系统结构如图 11-3 中圆圈区域所示，主要包含如下 4 个组件。

- AGU 主要为读和写指令，以及“A”扩展指令生成存储器访问地址。
- LSU 主要作为存储器访问的控制模块。
- ITCM 主要作为存储器子系统的指令存储部件，但是也能够用于存储数据而被读和写指令访问。
- DTCM 作为存储器子系统的数据存储部件。

下文将分别予以论述。

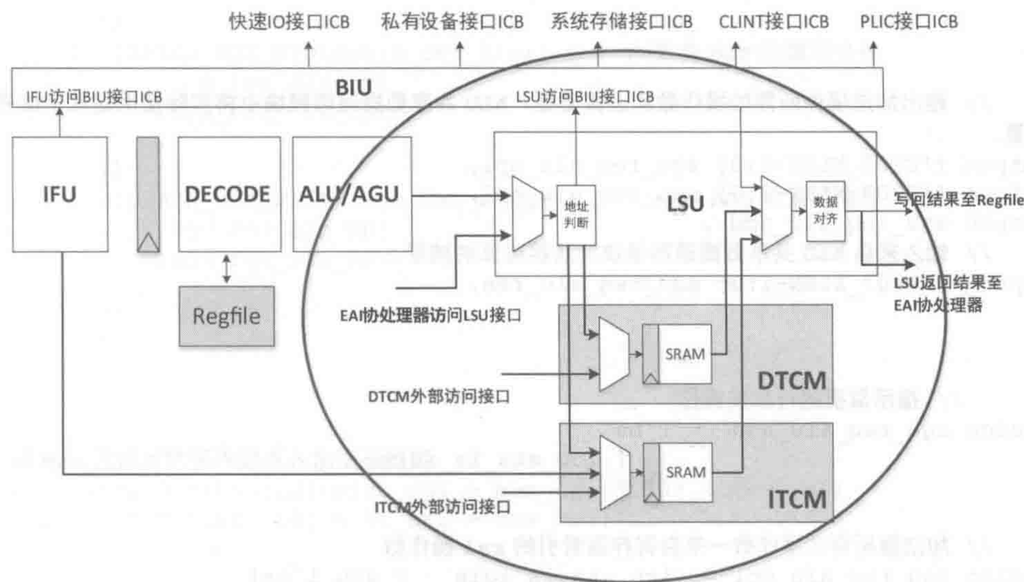


图 11-3 蜂鸟 E200 处理器核的存储器子系统结构示意图

11.4.2 AGU

AGU（Address Generation Unit）主要用于产生读/写指令以及“A”扩展指令的访存地址，其微架构的要点如下。

- 如图 11-3 所示，AGU 是 ALU 的一个子单元。AGU 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                              // 存放 RTL 的目录
        |----e203                         // E203 核和 SoC 的 RTL 目录
            |----core                     // 存放 e203 Core 的 RTL 代码
                |----e203_exu_alu.v            // ALU 顶层模块
                |----e203_exu_alu_dpath.v      // ALU 的数据通路（包含加法器）
                |----e203_exu_alu_lsuagu.v     // AGU 的源代码
```

- 根据 RISC-V 架构定义，读/写指令需要将其第一个寄存器索引的源操作数和符号位扩展的立即数相加，得到最终的访存地址，因此理论上需要使用到加法器。为了节省面积，蜂鸟 E200 处理器复用 ALU 的加法器用于访存地址计算。相关的源代码片段如下所示。

```
// e203_exu_alu_lsuagu.v 源代码片段

  // 此模块作为 AGU 的源代码模块，其中主要实现了相关的控制和选择，并没有实际使用加法器。

......

    // 输出加法操作所需的操作数和运算类型，ALU 共享数据通路模块中将实际使用这几个信号进行
加法运算。
  output ['E203_XLEN-1:0] agu_req_alu_op1,
  output ['E203_XLEN-1:0] agu_req_alu_op2,
  output agu_req_alu_add ,
    // 输入来自 ALU 共享数据通路模块加法器运算的结果。
  input  ['E203_XLEN-1:0] agu_req_alu_res,

......

      // 指示需要进行加法操作
  assign agu_req_alu_add  = 1'b0
                          | icb_sta_is_idle
                          ;

    // 加法器所需的操作数一来自寄存器索引的 rs1 操作数
  assign agu_req_alu_op1 =  icb_sta_is_idle   ? agu_i_rs1
                 : 'E203_XLEN'd0;

    // 加法器所需的操作数二来自立即数
```

```
    wire ['E203_XLEN-1:0] agu_addr_gen_op2 = agu_i_ofst0 ? 'E203_XLEN'b0 : ag
u_i_imm;
    assign agu_req_alu_op2 =  icb_sta_is_idle   ? agu_addr_gen_op2
                              : 'E203_XLEN'd0;

  ......

     // 使用来自 ALU 共享数据通路模块进行加法运算的结果，作为访存的地址信号
    assign agu_icb_cmd_addr = agu_req_alu_res['E203_ADDR_SIZE-1:0];
```

```
  // e203_exu_alu_dpath.v 源代码片段

    // 此模块作为 ALU 的共享数据通路，其中主要包含了一个加法器。ALU 所处理的所有指令的实际运
算均由此模块的数据通路执行。

  ......

    assign  { // 来自 ALU，BJP，AGU 模块的请求信号组成一个 Mux 共享 ALU 的操作数
      mux_op1
     ,mux_op2
    ......
     }
     =
        ({DPATH_MUX_WIDTH{alu_req_alu}} & { //来自 ALU 的请求信号
             alu_req_alu_op1
            ,alu_req_alu_op2
            ......
        })
      | ({DPATH_MUX_WIDTH{bjp_req_alu}} & { //来自 BJP 的请求信号
             bjp_req_alu_op1
            ,bjp_req_alu_op2
            ......
        })
      | ({DPATH_MUX_WIDTH{agu_req_alu}} & { //来自 AGU 的请求信号
             agu_req_alu_op1
            ,agu_req_alu_op2
           ......
        })
        ;

  ......

  //将 Mux 选择后的操作数送入加法器通路
     wire ['E203_XLEN-1:0] misc_op1 = mux_op1['E203_XLEN-1:0];
     wire ['E203_XLEN-1:0] misc_op2 = mux_op2['E203_XLEN-1:0];

  ......
     wire ['E203_ALU_ADDER_WIDTH-1:0] misc_adder_op1 =
        {{'E203_ALU_ADDER_WIDTH-'E203_XLEN{(~op_unsigned) & misc_op1['E203_XL
EN-1]}},misc_op1};
```

```
  wire ['E203_ALU_ADDER_WIDTH-1:0] misc_adder_op2 =
      {{'E203_ALU_ADDER_WIDTH-'E203_XLEN{(~op_unsigned) & misc_op2['E203_XL
EN-1]}},misc_op2};
 ......

//生成加法器的操作数
  wire ['E203_ALU_ADDER_WIDTH-1:0] adder_op1 =
'ifdef E203_SUPPORT_SHARE_MULDIV //{
      muldiv_req_alu ? muldiv_req_alu_op1 :
'endif//}
      misc_adder_op1;
  wire ['E203_ALU_ADDER_WIDTH-1:0] adder_op2 =
'ifdef E203_SUPPORT_SHARE_MULDIV //{
      muldiv_req_alu ? muldiv_req_alu_op2 :
'endif//}
      misc_adder_op2;

  wire adder_cin;
  wire ['E203_ALU_ADDER_WIDTH-1:0] adder_in1;
  wire ['E203_ALU_ADDER_WIDTH-1:0] adder_in2;
  wire ['E203_ALU_ADDER_WIDTH-1:0] adder_res;

//判断所需的是加法还是减法操作
  wire adder_add;
  wire adder_sub;

  assign adder_add =
'ifdef E203_SUPPORT_SHARE_MULDIV //{
      muldiv_req_alu ? muldiv_req_alu_add :
'endif//}
      op_add;
  assign adder_sub =
'ifdef E203_SUPPORT_SHARE_MULDIV //{
      muldiv_req_alu ? muldiv_req_alu_sub :
'endif//}
           (
               // The original sub instruction
           (op_sub)
               // The compare lt or gt instruction
          | (op_cmp_lt | op_cmp_gt |
            op_cmp_ltu | op_cmp_gtu |
            op_max | op_maxu |
            op_min | op_minu |
            op_slt | op_sltu
           ));

  wire adder_addsub = adder_add | adder_sub;

//假设当前的操作不是加法或者减法操作，则将加法器的输入门控（Gate）住以节省动态功耗
     // Make sure to use logic-gating to gate off the adder
  assign adder_in1 = {'E203_ALU_ADDER_WIDTH{adder_addsub}} & (adder_op1);
//使用取反加一的方式将补码减法转换成加法操作
```

```
    assign adder_in2 = {'E203_ALU_ADDER_WIDTH{adder_addsub}} & (adder_sub ? (
~adder_op2) : adder_op2);
    assign adder_cin = adder_addsub & adder_sub;

  //最终实际的加法器数据通路
    assign adder_res = adder_in1 + adder_in2 + adder_cin;

  //将 ALU 的加法器计算结果返回给 AGU
     assign agu_req_alu_res       = alu_dpath_res['E203_XLEN-1:0];

  ......
```

- RISC-V 架构对于地址非对齐（Address Misalign）的读和写指令，可以使用硬件支持，也可以使用软件通过异常服务程序的方式采用软件支持。蜂鸟 E200 与伯克利开源的 Rocket Core 一样选择采用软件支持。AGU 通过对生成出的访存地址进行判断，如果其地址非对齐，则产生异常标志。通过 ALU 传送给交付模块，交付模块则据此产生异常（请参见第 13.5 节了解更多异常处理的信息）。相关的源代码片段如下所示。

```
// e203_exu_alu_lsuagu.v 源代码片段
 ......

   // 判断当前读、写指令访问内存的操作尺寸（Size）
  wire agu_i_size_b  = (agu_i_size == 2'b00);
  wire agu_i_size_hw = (agu_i_size == 2'b01);
  wire agu_i_size_w  = (agu_i_size == 2'b10);

  //将 ALU 的加法器计算结果作为读、写指令访存的地址
  assign agu_icb_cmd_addr = agu_req_alu_res['E200_ADDR_SIZE-1:0];

   // 判断当前访存的地址是否和操作尺寸对齐（Memory Aligned）
  wire agu_i_addr_unalgn =
          //如果地址最低位不为 0，意味着和半字（Half-Word）不对齐
          (agu_i_size_hw &  agu_icb_cmd_addr[0])
            //如果地址最低两位不为 0，意味着和字（Word）不对齐
        | (agu_i_size_w  &  (|agu_icb_cmd_addr[1:0]));

  wire state_last_exit_ena;

  wire agu_addr_unalgn = agu_i_addr_unalgn;

  wire agu_i_unalgnld = agu_addr_unalgn & agu_i_load;
  wire agu_i_unalgnst = agu_addr_unalgn & agu_i_store;

   // 产生非对齐指示信号给交付接口
  assign agu_o_cmt_misalgn = 1'b0
                     | (agu_i_unalgnldst) ;
   // 产生 Load 指令指示信号给交付接口，将
```

```
    // 用于产生读存储器地址不对齐异常（Load address misaligned Exception）
  assign agu_o_cmt_ld        = agu_i_load & (~agu_i_excl);
    // 产生 Store 和 AMO 指令指示信号给交付接口，将
    // 用于产生写存储器或 AMO 地址不对齐异常（Store/AMO address misaligned Exception）
  assign agu_o_cmt_stamo     = agu_i_store | agu_i_amo | agu_i_excl;
```

- 如果没有产生异常的读和写指令，则通过 AGU 的 ICB 接口发送给 LSU 模块，如图 11-3 所示。有关 ICB 接口协议的详细介绍，请参见第 12.2 节。注意：如果产生了异常的读、写指令，则不会发送给 LSU 模块。相关源代码片段如下所示。

```
// e203_exu_alu_lsuagu.v 源代码片段

    // 产生 ICB 接口的 cmd_valid 信号
  assign agu_icb_cmd_valid =
           // 只有地址对齐（不会产生异常） 的指令才会生成 cmd_valid
          ((agu_i_algnldst & agu_i_valid)
    // 为了保证指令被同时发送给交付接口和 ICB 接口，必须“与”上交付接
      // 口的接收 ready 信号“agu_o_ready”
            & (agu_o_ready)
          )
          ;

      // 产生 ICB 接口的 cmd_addr 信号（使用 ALU 的加法器计算结果）和 cmd_read 信号
    assign agu_icb_cmd_addr = agu_req_alu_res['E203_ADDR_SIZE-1:0];
    assign agu_icb_cmd_read =
          (agu_i_algnldst & agu_i_load)
        ;

      // 产生 ICB 接口的 cmd_wdata 和 cmd_wmask 信号，由于需经过 32 位宽的 ICB 总线，所以
必须经过操作尺寸（Size）对齐
    wire ['E203_XLEN-1:0] algnst_wdata =
          ({'E203_XLEN{agu_i_size_b }} & {4{agu_i_rs2[ 7:0]}})
        | ({'E203_XLEN{agu_i_size_hw}} & {2{agu_i_rs2[15:0]}})
        | ({'E203_XLEN{agu_i_size_w }} & {1{agu_i_rs2[31:0]}});

    wire ['E203_XLEN/8-1:0] algnst_wmask =
          ({'E203_XLEN/8{agu_i_size_b }} & (4'b0001 << agu_icb_cmd_addr[1:0]))
        | ({'E203_XLEN/8{agu_i_size_hw}} & (4'b0011 << {agu_icb_cmd_addr[1]
,1'b0}))
        | ({'E203_XLEN/8{agu_i_size_w }} & (4'b1111));


    assign agu_icb_cmd_wdata = algnst_wdata;
    assign agu_icb_cmd_wmask = algnst_wmask;
```

11.4.3 LSU

LSU 全称为 Load Store Unit，是蜂鸟 E200 处理器核存储器子系统的主要控制单元，如图 11-3 所示，其要点如下。

- LSU 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                               // 存放RTL的目录
         |----e203                         // E203核和SoC的RTL目录
              |----core                    // 存放e203 Core的RTL代码
                   |----e203_lsu.v                  // LSU顶层模块
                   |----e203_lsu_ctrl.v             // LSU模块主体控制
```

- LSU 的微架构和代码实现与 BIU 模块非常相似，建议读者先阅读第 12 章了解有关 ICB 和 BIU 的介绍，更有利于理解本节内容。
- LSU 有两组输入 ICB 总线接口，分别来自于 AGU 单元和 EAI 协处理器（有关 EAI 协处理器接口，请参见第 16.3 节）。有 3 组输出 ICB 总线接口，分别分发给 BIU、DTCM 和 ITCM。另外，LSU 通过其写回接口将结果写回。
- 两组输入 ICB 总线经过一个“ICB 汇合”模块将其汇合成为一组 ICB 总线，采用的仲裁机制是优先级仲裁，EAI 总线具有更高的优先级。有关“ICB 汇合”模块的详细介绍，请参见第 12.3.2 节。
- 经过汇合之后的 ICB 总线通过其命令通道（Command Channel）的地址进行判断，通过其访问的地址区间产生分发信息，然后使用一个“ICB 分发”模块将其分发给不同的存储器组件的 ICB 接口（包括 BIU，DTCM 和 ITCM）。有关“ICB 分发”模块的详细介绍，请参见第 12.3.1 节。
- LSU 中使用到的“ICB 汇合”和“ICB 分发”模块的 FIFO 深度默认配置均为 1，意味着蜂鸟 E200 处理器的 LSU 默认只支持一个滞外交易（One Outstanding Transaction），此配置的原因在于减少面积开销。
- 最终的返回数据经过操作尺寸（Size）对齐之后，经过 LSU 的写回接口写回。其相关源代码片段如下所示。

```
// e203_lsu_ctrl.v 源代码片段

  wire ['E203_XLEN-1:0] rdata_algn =
      (pre_agu_icb_rsp_rdata >> {pre_agu_icb_rsp_addr[1:0],3'b0});

  wire rsp_lbu = (pre_agu_icb_rsp_size == 2'b00) & (pre_agu_icb_rsp_usign =
= 1'b1);
  wire rsp_lb  = (pre_agu_icb_rsp_size == 2'b00) & (pre_agu_icb_rsp_usign =
= 1'b0);
  wire rsp_lhu = (pre_agu_icb_rsp_size == 2'b01) & (pre_agu_icb_rsp_usign =
= 1'b1);
  wire rsp_lh  = (pre_agu_icb_rsp_size == 2'b01) & (pre_agu_icb_rsp_usign =
= 1'b0);
  wire rsp_lw  = (pre_agu_icb_rsp_size == 2'b10);
```

```
    wire ['E203_XLEN-1:0] sc_excl_wdata = pre_agu_icb_rsp_excl_ok ? 'E203_XLE
N'd0 : 'E203_XLEN'd1;

  // 对返回的数据进行符号扩展和操作尺寸对齐
    assign lsu_o_wbck_wdat   = pre_agu_icb_rsp_excl ? sc_excl_wdata :
           ( ({'E203_XLEN{rsp_lbu}} & {{24{          1'b0}}, rdata_algn[ 7:0]})
           | ({'E203_XLEN{rsp_lb }} & {{24{rdata_algn[ 7]}}, rdata_algn[ 7:0]})
           | ({'E203_XLEN{rsp_lhu}} & {{16{          1'b0}}, rdata_algn[15:0]})
           | ({'E203_XLEN{rsp_lh }} & {{16{rdata_algn[15]}}, rdata_algn[15:0]})
           | ({'E203_XLEN{rsp_lw }} & rdata_algn[31:0]));
```

- 由于访问不同的存储器组件（包括 BIU、DTCM 和 ITCM）可能会出现存储器访问错误（Memory Access Fault），可以通过 ICB 的反馈通道（Response Channel）返回标志信号所得。如果出现此错误，则产生异常标志通过 LSU 的写回接口传送给交付模块，交付模块则据此产生异常（请参见第 13.5 节了解更多异常处理的信息）。相关的源代码片段如下所示。

```
  // e203_lsu_ctrl.v 源代码片段

     // 产生存储器访问错误指示信号给交付接口
    assign lsu_o_cmt_buserr  = pre_agu_icb_rsp_err;// The bus-error exception
 generated

     // 出现存储器访问错误的访存地址
    assign lsu_o_cmt_badaddr = pre_agu_icb_rsp_addr;

  // 产生 Load 指令指示信号给交付接口，将用于产生读存储器错误异常
（Load Access Fault Exception）
    assign lsu_o_cmt_ld=  pre_agu_icb_rsp_read;

  // 产生 Store 指令指示信号给交付接口，将用于产生写存储器或 AMO 错误异常
（Store/AMO Access Fault Exception）
    assign lsu_o_cmt_st= ~pre_agu_icb_rsp_read;
```

11.4.4 ITCM 和 DTCM

第 11.1.3 节讨论了 ITCM 和 DTCM 在嵌入式低功耗处理器中的优点，如图 11-3 所示，蜂鸟 E200 即配备了专用的 ITCM（数据宽度为 64 位）和 DTCM（数据宽度为 32 位）。

当今哈佛结构和冯•诺依曼结构的严格界限已经变得模糊，蜂鸟 E200 处理器核的实现同样如此。

- 蜂鸟 E200 处理器有专用的总线分别访问 ITCM 和 DTCM，因此从此方面来看，蜂鸟 E200 处理器核属于哈佛结构。
- 如图 11-3 所示，ITCM 也有一组输入 ICB 总线接口来自 LSU 的访问，也就是说 ITCM

所在的地址区间同样能够通过 LSU 被读和写指令访问到用于存储数据。因此从ITCM 的角度来看，蜂鸟 E200 处理器又可以认为属于冯•诺依曼结构。

注意：读和写指令对于 ITCM 的访问主要用于程序的上电初始化（譬如从 Flash 中将程序读出并写入 ITCM 中）。在程序正常运行时，不推荐将数据段放入 ITCM，否则性能无法得到最大化利用。

ITCM 的微架构如图 11-4 所示，要点如下。

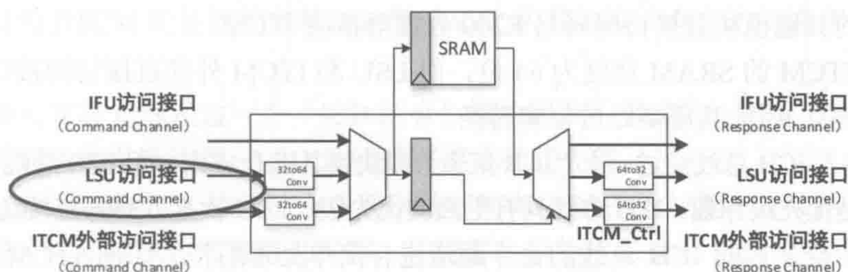


图 11-4 ITCM 微架构示意图

- ITCM 的存储器主体由一块数据宽度为 64 位的单口 SRAM 组成。ITCM 的大小和基地址（位于全局地址空间中的起始地址）可以通过 config.v 中的宏定义参数配置。请参见第 4.6 节了解更多可配置的信息。

```
e200_opensource
   |----rtl                          // 存放 RTL 的目录
       |----e203                     // E203 核和 SoC 的 RTL 目录
           |----core                 // 存放 e203 Core 的 RTL 代码
               |----config.v         // 配置文件
```

- ITCM 采用数据宽度为 64 位能够取得更低的功耗消耗。

  首先，使用宽度为 64 位的 SRAM 在物理大小上比 32 位的 SRAM 面积更加紧凑，因此同样容量 ITCM 使用 64 位的数据宽度比使用 32 位的数据宽度面积更小。

  其次，程序在执行的过程中大多数是顺序取指令，而 64 位宽的 ITCM 可以一次取出 64 位的指令流，相比于从 32 位宽的 ITCM 中连续读两次取出 64 位的指令流，只读一次 64 位宽的 SRAM 能够消耗更少的动态功耗。请参见第 7.3 节了解更多取指部分的实现细节。

- ITCM 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
   |----rtl                          // 存放 RTL 的目录
       |----e203                     // E203 核和 SoC 的 RTL 目录
           |----core                 // 存放 e203 Core 的 RTL 代码
               |----e203_itcm_ctrl.v           // ICTM 模块主体控制
```

- 值得再次强调的是，ITCM有一组输入ICB总线接口（数据宽度为32位）来自LSU的访问，如图11-4中圆圈所示。也就是说ITCM所在的地址区间同样能够通过LSU被Load和Store指令访问到，从而可以用于存储数据。
- ITCM还有另外两组输入ICB总线接口，数据宽度为64位的IFU专用ICB接口和数据宽度为32位的外部直接访问接口（ITCM External ICB Interface）。ITCM外部直接访问接口（ITCM External ICB Interface）是专门为ITCM配备的外部接口，便于SoC的其他模块直接访问蜂鸟E200处理器核的ITCM。
- 由于ITCM的SRAM宽度为64位，而LSU和ITCM外部直接访问接口的数据宽度为32位，因此其需要经过位宽转换。
- 3组输入ICB总线经过一个"ICB汇合"模块将其汇合成为一组ICB总线，采用的仲裁机制是优先级仲裁。IFU总线具有更高的优先级、LSU次之、外部直接访问接口最低。
- 经过汇合之后的ICB总线的命令通道进行简单处理后作为访问ITCM SRAM的接口。同时将此操作的来源信息寄存，并用寄存后的信息指示SRAM返回的数据分发给IFU、LSU和ITCM外部直接访问接口的反馈通道。
- 由于ITCM控制模块的源代码比较简单，请读者于GitHub上的e200_opensource项目中自行阅读。

DTCM的微架构如图11-5所示，要点如下。

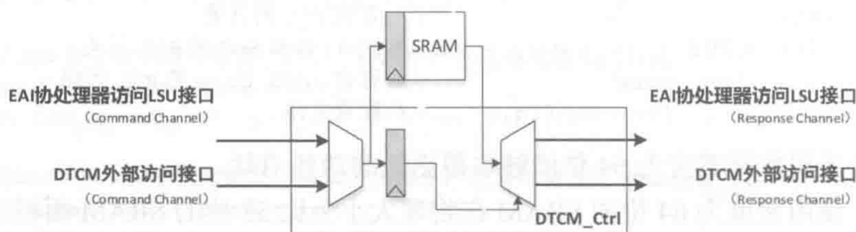


图11-5 DTCM微架构示意图

- DTCM的存储器主体由一块数据宽度为32位的单口SRAM组成。DTCM的大小和基地址（位于全局地址空间中的起始地址）可以通过config.v中的宏定义参数配置。请参见第4.6节了解更多可配置的信息。

```
e200_opensource
    |----rtl                              // 存放RTL的目录
        |----e203                         // E203核和SoC的RTL目录
            |----core                     // 存放e203 Core的RTL代码
                |----config.v             // 配置文件
```

- DTCM的相关源代码在e200_opensource目录的结构如下。关于GitHub网站上e200_opensource开源项目的完整代码层次结构详解，请参见第17.1节。

```
e200_opensource
    |----rtl                         // 存放 RTL 的目录
        |----e203                    // E203 核和 SoC 的 RTL 目录
            |----core                // 存放 e203 Core 的 RTL 代码
                |----e203_dtcm_ctrl.v          // DCTM 模块主体控制
```

- DTCM 有两组输入 ICB 总线接口，分别来自于 LSU 和外部直接访问接口（DTCM External ICB Interface）。DTCM 外部直接访问接口（DTCM External ICB Interface）是专门为 DTCM 配备的外部接口，便于 SoC 的其他模块直接访问蜂鸟 E200 处理器核的 DTCM。
- 两组输入 ICB 总线经过一个“ICB 汇合”模块将其汇合成为一组 ICB 总线，采用的仲裁机制是优先级仲裁，LSU 总线具有更高的优先级。
- 经过汇合之后的 ICB 总线的命令通道进行简单处理后作为访问 DTCM SRAM 的接口。同时将此操作的来源信息寄存，并用寄存后的信息指示 SRAM 返回的数据分发给 LSU 和 DTCM 外部直接访问接口的反馈通道。
- 由于 DTCM 控制模块的源代码比较简单，请读者于 GitHub 上的 e200_opensource 项目中自行阅读。

1.4.5 “A”扩展指令处理

虽然“A”扩展子集对于蜂鸟 E200 这样的极低功耗处理器而言未必是必须要支持的一部分，但是由于 RV32IMAC 架构组合是目前比较主流的工具链支持版本，因此开源的蜂鸟 E203 处理器核选择默认支持“A”扩展子集。

如第 11.3.3 节所述，“A”扩展指令分为两类，蜂鸟 E200 处理器对其的硬件实现以下予以分别论述。

1. Load-Reserved 和 Store-Conditional 指令的硬件实现

为了能够支持 RISC-V 架构中定义的 Load-Reserved 和 Store-Conditional 指令的行为，蜂鸟 E200 在 LSU 单元中设置了一个互斥检测器(Exclusive Monitor)，该互斥检测器的行为如下。

- 当有一个 Load-Reserved 指令执行时，设置互斥检测器的有效标志，并将互斥检测器中存入该指令访问存储器的地址。
- 当有任何一个写指令（包括普通的写或者 Store-Conditional）执行时，如果写指令访问存储器的地址和互斥检测器中存储的地址一样，则将互斥检测器的有效标志清除掉。
- 如果发生了任何的异常和中断或者执行了 mret 指令，也会将互斥检测器的有效标志清除掉。
- 当有一条 Store-Conditional 指令执行时，如果互斥检测器里的有效位为高，且其中保存的地址值和该 Store-Conditional 指令访问存储器的地址值相同，则意味着该

Store-Conditional 指令能够执行成功，否则执行失败。

如果 Store-Conditional 执行成功，则会真正向存储器中写入数值，且向结果寄存器中写回的结果为 0；如果该 Store-Conditional 执行失败，则会放弃写入存储器，并且向结果寄存器中写回结果为 1。

- 注意：理论上 RISC-V 的 Load-Reserved 和 Store-Conditional 指令可以支持获取（acquire）与释放（release）属性。由于蜂鸟 E200 中存储器访问指令严格按顺序执行，因此等效于永远将 Load-Reserved 和 Store-Conditional 指令当作同时具备获取与释放属性来实现。有关获取与释放属性请参见附录 A14.5 节了解 Load-Reserved 和 Store-Conditional 指令细节。
- 注意：由于该互斥检测器（Exclusive Monitor）存在于蜂鸟 E200 的内部，因此它仅跟踪记录了蜂鸟 E200 单核对于存储器空间的访问。但是在多核系统中，假设其他的核或者模块也访问了相同的地址空间，则无法被检测到。因此蜂鸟 E200 的该 Exclusive Monitor 实现只能够在单核独自访问存储器时保证程序执行 Load-Reserved 和 Store-Conditional 指令的结果正确。而当有多个核或者其他模块（譬如 DMA）同时也在访问互斥检测器记录的地址时，则可能会造成 Load-Reserved 和 Store-Conditional 指令的结果无法准确反映。

  之所以存在此局限性，是因为蜂鸟 E200 支持“A”扩展子集的意图，主要是为了使其能够使用最常见的 RV32IMAC 工具链，而并非为了支持严格的多核功能。同时蜂鸟 E200 的开发者认为，蜂鸟 E200 此种类型的超低功耗处理器核的大多数应用场景应为单核场景。

以上功能在 LSU 模块中的相关源代码片段如下所示。

```
    // e203_lsu_ctrl.v 源代码片段


    wire excl_flg_r;
    wire ['E203_ADDR_SIZE-1:0] excl_addr_r;
    wire icb_cmdaddr_eq_excladdr = (arbt_icb_cmd_addr == excl_addr_r);

  //当有一个 Load-Reserved 指令执行之时，将互斥检测器的有效标志设置起来。
    wire excl_flg_set = splt_fifo_wen & arbt_icb_cmd_usr[USR_PACK_EXCL] & arb
t_icb_cmd_read & arbt_icb_cmd_excl;
  //当有任何一个写指令执行之时，如果写指令的访存地址和互斥检测器中存储的有效地址一样，则将互
斥检测器的有效标志清除掉。
    wire excl_flg_clr = (splt_fifo_wen & (~arbt_icb_cmd_read) & icb_cmdaddr_e
q_excladdr & excl_flg_r)
      //并且如果发生了任何的异常和中断或者执行了 mret 指令，也会将互斥检测器的有效标志清除掉。
                         | commit_trap | commit_mret;
    wire excl_flg_ena = excl_flg_set | excl_flg_clr;
    wire excl_flg_nxt = excl_flg_set | (~excl_flg_clr);
    sirv_gnrl_dfflr #(1) excl_flg_dffl (excl_flg_ena, excl_flg_nxt, excl_flg_
```

```
r, clk, rst_n);

    //当有一个 Load-Reserved 指令执行之时，将互斥检测器的有效标志设置起来时，也将互斥检测器
中存入该指令访问存储器的地址。
      wire excl_addr_ena = excl_flg_set;
      wire ['E203_ADDR_SIZE-1:0] excl_addr_nxt = arbt_icb_cmd_addr;
      sirv_gnrl_dfflr #('E203_ADDR_SIZE) excl_addr_dffl (excl_addr_ena, excl_ad
dr_nxt, excl_addr_r, clk, rst_n);

      //判断 Store-Conditional 指令是否能够执行成功
            //如果 Store-Conditional 指令执行时，互斥检测器里的有效位为高，且其中保存的地址
值和 Store-Conditional 指令访问存储器的地址值相同，则执行成功。
      wire arbt_icb_cmd_scond_true = arbt_icb_cmd_scond & icb_cmdaddr_eq_exclad
dr & excl_flg_r;


    //如果 Store-Conditional 指令不能够执行成功，则将 ICB 总线命令通道的 Write-Mask 信号设置
为 0，这样就不会真正向存储器中写入数值。
      wire ['E203_XLEN/8-1:0] arbt_icb_cmd_wmask_pos =
          (arbt_icb_cmd_scond & (~arbt_icb_cmd_scond_true))
            ? {'E203_XLEN/8{1'b0}} : arbt_icb_cmd_wmask;

    //如果 Store-Conditional 指令不能够执行成功，则向结果寄存器写回的值是 0，否则写回的值是 1。
      wire ['E203_XLEN-1:0] sc_excl_wdata = arbt_icb_rsp_scond_true ? 'E203_XLE
N'd0 : 'E203_XLEN'd1;
```

2. AMO（Atomic Memory Operation）指令的硬件实现

为了能够支持 RISC-V 架构中定义的 AMO 指令的行为，蜂鸟 E200 在 AGU 单元中使用了状态机将 AMO 指令拆分为若干个不同的微操作，其步骤分别为如下。

- 首先拆分出一个存储器读操作，等待其读操作的数据返回，并将返回的数据寄存。
- 然后将“寄存的读操作返回数据”进行相应的算术运算，将运算的结果寄存在电路中。
- 再次发起一个存储器写操作将“寄存的运算结果”写入存储器，等待其反馈返回（确定写操作是否成功，是否发生存储器访问错误）。
- 最后，如果在上述过程中没有发生存储器访问错误，则将“寄存的读操作返回数据”写回该指令的结果寄存器。
- 注意：理论上 RISC-V 的 AMO 指令可以支持获取与释放属性。由于蜂鸟 E200 中存储器访问指令严格按顺序执行，因此等效于永远将 AMO 指令当作同时具备获取与释放属性来实现。有关获取与释放属性请参见附录 A14.5 节了解 AMO 指令细节。
- 注意：由于蜂鸟 E200 需要在总线上先后进行两次操作，拆分出如上这些微操作，第一次是读操作，第二次是写操作，在此期间并未将外部总线锁定。因此在多核系统中，假设其他的核或者模块也访问了相同的地址空间，则破坏了其原子性。蜂鸟 E200 的实现只能够在单核独自访问存储器时保证程序执行 AMO 指令的结果正确；而当有多个核或者其他模块（譬如 DMA）同时也在访问 AMO 指令的地址时，则会造成

AMO 指令执行，无法真正实现原子性。

之所以存在此局限性，是因为蜂鸟 E200 支持“A”扩展子集的意图主要是为了使其能够使用最常见的 RV32IMAC 工具链，而并非为了支持严格的多核功能。同时蜂鸟 E200 的开发者认为，蜂鸟 E200 此种类型的超低功耗处理器核的大多数应用场景应为单核场景。

以上功能在 AGU 模块中的相关源代码片段如下所示。

```
// e203_exu_alu_lsuagu.v 源代码片段

// 此模块作为 AGU 的源代码模块，其中主要实现了相关的控制和选择，并没有包含实际的运算数据通路，真实的数据通路共享 ALU 的运算数据通路。


//以下状态机控制 AMO 指令的拆分。
   localparam ICB_STATE_WIDTH = 4;

   wire icb_state_ena;
   wire [ICB_STATE_WIDTH-1:0] icb_state_nxt;
   wire [ICB_STATE_WIDTH-1:0] icb_state_r;
   // 空闲状态，该状态下可以开始发送第一次读操作
   localparam ICB_STATE_IDLE = 4'd0;

   // 已经发送了第一次读操作，等待读数据返回的状态
   localparam ICB_STATE_1ST  = 4'd1;

   // 发送第二次写操作的状态
   localparam ICB_STATE_WAIT2ND  = 4'd2;

   // 已经发送了第二次写操作，等待反馈返回的状态
   localparam ICB_STATE_2ND  = 4'd3;

   // 收到第一次读操作返回的读数据后，复用 ALU 的运算数据通路进行运算的状态
   localparam ICB_STATE_AMOALU  = 4'd4;

   //进行运算后的结果已经寄存在电路中准备好，并正在发送写操作的状态
   localparam ICB_STATE_AMORDY  = 4'd5;

   //写操作的反馈已经返回，将指令的结果写回结果寄存器中的状态
   localparam ICB_STATE_WBCK  = 4'd6;


   wire [ICB_STATE_WIDTH-1:0] state_idle_nxt   ;
   wire [ICB_STATE_WIDTH-1:0] state_1st_nxt    ;
   wire [ICB_STATE_WIDTH-1:0] state_wait2nd_nxt;
   wire [ICB_STATE_WIDTH-1:0] state_2nd_nxt    ;
   wire [ICB_STATE_WIDTH-1:0] state_amoalu_nxt ;
   wire [ICB_STATE_WIDTH-1:0] state_amordy_nxt ;
   wire [ICB_STATE_WIDTH-1:0] state_wbck_nxt ;
```

```
wire state_1st_exit_ena      ;
wire state_wait2nd_exit_ena  ;
wire state_2nd_exit_ena      ;
wire state_amoalu_exit_ena   ;
wire state_amordy_exit_ena   ;
wire state_wbck_exit_ena   ;

wire  icb_sta_is_idle     = (icb_state_r == ICB_STATE_IDLE    );
wire  icb_sta_is_1st      = (icb_state_r == ICB_STATE_1ST     );
wire  icb_sta_is_amoalu   = (icb_state_r == ICB_STATE_AMOALU );
wire  icb_sta_is_amordy   = (icb_state_r == ICB_STATE_AMORDY );
wire  icb_sta_is_wait2nd = (icb_state_r == ICB_STATE_WAIT2ND);
wire  icb_sta_is_2nd      = (icb_state_r == ICB_STATE_2ND     );
wire  icb_sta_is_wbck     = (icb_state_r == ICB_STATE_WBCK     );
```

......

```
//状态机的状态寄存器。
  sirv_gnrl_dfflr #(ICB_STATE_WIDTH) icb_state_dfflr (icb_state_ena, icb_st
ate_nxt, icb_state_r, clk, rst_n);

//寄存第一次读操作返回的数据，但是此模块并没有实际例化寄存器，而是使用 ALU 的数据通路模块中
的寄存器（与多周期乘除法器复用）。
  assign leftover_ena = agu_icb_rsp_hsked & (
                 1'b0
                 'ifdef E203_SUPPORT_AMO//{
                 | amo_1stuop
                 | amo_2nduop
                 'endif//E203_SUPPORT_AMO}
                 );
  assign leftover_nxt =
            {'E203_XLEN{1'b0}}
        'ifdef E203_SUPPORT_AMO//{
          | ({'E203_XLEN{amo_1stuop}} & agu_icb_rsp_rdata)
          | ({'E203_XLEN{amo_2nduop}} & leftover_r)
        'endif//E203_SUPPORT_AMO}
          ;

//寄存算术运算的结果，但是此模块并没有实际例化寄存器，而是使用 ALU 的数据通路模块中的寄存器
（与多周期乘除法器复用）。
  assign leftover_1_ena = 1'b0
        'ifdef E203_SUPPORT_AMO//{
          | icb_sta_is_amoalu
        'endif//E203_SUPPORT_AMO}
        ;

  assign leftover_1_nxt = agu_req_alu_res;

//向 ALU 共享的运算数据通路发送操作数
```

```
    assign agu_req_alu_op1 = ......
    assign agu_req_alu_op2 =  ......

//向 ALU 共享的运算数据通路发送具体的运算类型,
      assign agu_req_alu_add  = ......
    assign agu_req_alu_swap = (icb_sta_is_amoalu & agu_i_amoswap );
    assign agu_req_alu_and  = (icb_sta_is_amoalu & agu_i_amoand );
    assign agu_req_alu_or   = (icb_sta_is_amoalu & agu_i_amoor   );
    assign agu_req_alu_xor  = (icb_sta_is_amoalu & agu_i_amoxor );
    assign agu_req_alu_max  = (icb_sta_is_amoalu & agu_i_amomax );
    assign agu_req_alu_min  = (icb_sta_is_amoalu & agu_i_amomin );
    assign agu_req_alu_maxu = (icb_sta_is_amoalu & agu_i_amomaxu );
    assign agu_req_alu_minu = (icb_sta_is_amoalu & agu_i_amominu );
```

11.4.6 Fence 与 Fence.I 指令处理

在第 11.3.2 节中介绍了 Fence 和 Fence.I 指令的功能，蜂鸟 E200 处理器核对其的硬件实现要点如下。

- 理论上 RISC-V 的 Fence 指令可以区分 IORW 属性。蜂鸟 E200 永远将 Fence 指令当作"fence iorw, iorw"来实现。有关 IORW 属性请参见附录 A14.2 节了解 Fence 指令细节。
- 在流水线的派遣点，派遣 Fence 和 Fence.I 指令之前必须等待所有已经滞外的指令均执行完毕。这是一种最为简单的实现方案，较适合蜂鸟 E200 这样级别的处理器核。只需等到所有滞外指令均执行完毕，也就意味着所有的访存操作均已经完成，能够达到 Fence 和 Fence.I 指令需要分隔其前后指令访存操作的效果。相关源代码在 e200_opensource 目录的结构，以及相关的源代码片段如下所示。

```
e200_opensource
     |----rtl                              // 存放 RTL 的目录
          |----e203                         // E203 核和 SoC 的 RTL 目录
               |----core                     // 存放 e203 Core 的 RTL 代码
                    |----e203_exu_disp.v          // 指令派遣控制模块
```

```
    // e203_exu_disp.v 源代码片段

    // 判断当前指令是 Fence 或者 Fence.I 指令
    wire disp_fence_fencei    = (disp_i_info_grp == 'E203_DECINFO_GRP_BJP) &
                          ( disp_i_info ['E203_DECINFO_BJP_FENCE] | disp_i
_info ['E203_DECINFO_BJP_FENCEI]);

    // 派遣指令的条件信号，如果当前是 Fence 或者 Fence.I 指令，则必须等待 OITF 为空，OITF 记
录了所有的滞外指令，如果它为空则意味着所有滞外指令已经完成。
```

```
  wire disp_condition =
                ......
                  // To handle the Fence: just stall dispatch until the OITF
is empty
                & (disp_fence_fencei ? oitf_empty : 1'b1)
                ......
```

- 如第 11.3.2 节中所述，对于 Fence.I 指令而言，需要保证在 Fence.I 之后执行的取指令操作一定能够观测到在 Fence.I 指令之前执行的指令访存结果。Fence.I 指令在蜂鸟 E200 处理器核中将其被当作一种特殊的流水线冲刷（Pipeline Flush）指令来执行，其硬件实现与第 9 章中描述的分支指令解析一样，在 e203_exu_branchslv 模块中处理。相关源代码在 e200_opensource 目录的结构中，部分相关源代码片段如下所示。

```
e200_opensource
    |----rtl                              // 存放 RTL 的目录
        |----e203                          // E203 核和 SoC 的 RTL 目录
            |----core                        // 存放 e203 Core 的 RTL 代码
                |----e203_exu_commit.v          // 交付模块顶层
                |----e203_exu_branchslv.v// 交付模块中处理 Fence.I 指令的子模块
```

```
    // e200_exu_branchslv.v 源代码片段

// 生成流水线冲刷请求（Pipeline Flush Request），包括了 Fence.I 指令。

  wire brchmis_need_flush = (
       (cmt_i_bjp & (cmt_i_bjp_prdt ^ cmt_i_bjp_rslv))
  //   If it is a FenceI instruction, it is always Flush
       | cmt_i_fencei
  //   If it is a RET instruction, it is always jump
       | cmt_i_mret
  //   If it is a DRET instruction, it is always jump
       | cmt_i_dret
      );

// 如果是 Fence.I 指令，则造成流水线冲刷。使用 Fence.I 指令接下来的一条指令的 PC 作为冲刷请
求 PC（Flush PC），意味着 Fence.I 指令之后的指令流会被重新取指执行一遍，则达到了 Fence.I 指令
“需要保证在 Fence.I 之后执行的取指令操作一定能够观测到在 Fence.I 指令之前执行指令的访存结果”的
效果。

    assign brchmis_flush_pc =
  ......
                              (cmt_i_fencei | (cmt_i_bjp & cmt_i_bjp_prdt)) ? (
cmt_i_pc + (cmt_i_rv32 ? 'E200_PC_SIZE'd4 : 'E200_PC_SIZE'd2))
  ......
```

11.4.7 BIU

除了 ITCM 和 DTCM 之外，蜂鸟 E200 处理器核还能通过 BIU（Bus Interface Unit）访问外部的存储器，请参见第 12.4 节了解关于 BIU 的更多信息。

11.4.8 ECC

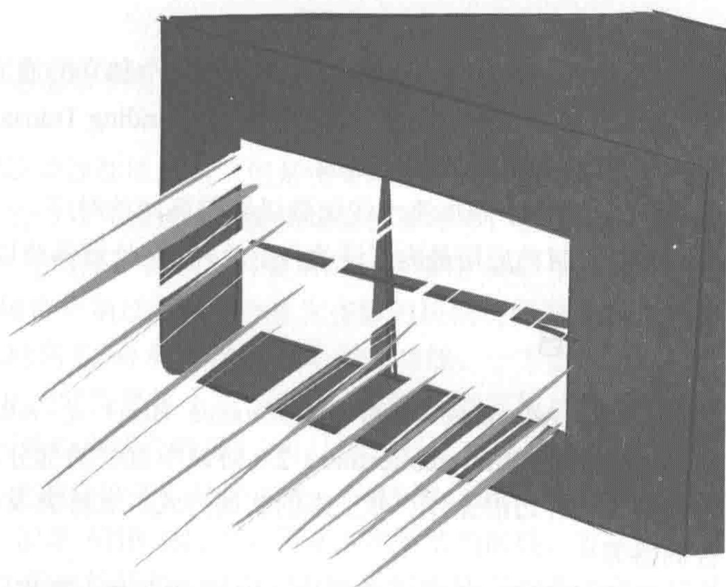
由于嵌入在处理器中的 SRAM 容易在极端情况下受到外界电离辐射的影响，从而使其保存的比特位发生翻转，使得 SRAM 中的数值发生改变造成错误。因此在很多高可靠性嵌入式处理器中，均使用 ECC（Error Checking and Correction）算法对 SRAM 进行保护，ECC 保护后的 SRAM 可以自动发现 1 位错误并自动纠正，而对 2 位错误可以发现此错误并将之上报系统。

蜂鸟 E200 处理器核也可以配置 ECC 算法，对 ITCM 和 DTCM 的 SRAM 进行保护，但是由于开源的 E200 处理器核并不提供此配置选项，因此本书在此不做赘述。

11.4.9 小结

蜂鸟 E200 处理器核使用了两级流水线的超低功耗架构（与 Cortex-M0+ 类似），配备了 ITCM 和 DTCM 及其专用的访问通道（与 Cortex-M3 类似），并配备了多样的外部存储器访问接口（与 Cortex-M4 类似）。可以说是集合了超低功耗处理器的优点于一身，是一款精心设计的处理器核。

第 12 章　黑盒子的窗口
——总线接口单元 BIU



对于一般处理器核的使用者，譬如软件人员或者 SoC 集成人员而言，可以将处理器核当作一个黑盒子，从而也许不用过于关心处理器核的内部运行，但是却一定需要了解并使用处理器核这个黑盒子的对外窗口，这个窗口便是总线接口单元（Bus Interface Unit，BIU）。

本章将介绍蜂鸟 E200 处理器核的 BIU 模块，介绍其使用的接口协议 ICB（Internal Chip Bus）以及 BIU 模块的微架构和源码分析。

12.1 片上总线协议概述

在介绍蜂鸟 E200 处理器核的总线接口之前，本节先简述几种常见的片上总线。

12.1.1 AXI

AXI（Advanced eXtensible Interface）是一种总线协议，是 ARM 公司提出的 AMBA（Advanced Microcontroller Bus Architecture）3.0 协议中最重要的部分，是一种面向高性能、高带宽、低延迟的片内总线。它有如下特点。

- 分离的地址和数据阶段。
- 支持地址非对齐的数据访问，使用字节掩码（Byte Strobes）来控制部分写操作。
- 使用基于突发的交易类型（Burst-based Transaction），对于突发操作仅需要发送起始地址，即可传输大片的数据。
- 分离的读通道和写通道，总共有 5 个独立的通道。
- 支持多个滞外交易（Multiple Outstanding Transaction）。
- 支持乱序返回乱序完成。
- 非常易于添加流水线级数以获得高频的时序。

AXI 是目前应用最为广泛的片上总线，是处理器核以及高性能 SoC 片上总线的事实标准。

12.1.2 AHB

AHB（Advanced High Performance Bus）是 ARM 公司提出的 AMBA（Advanced Microcontroller Bus Architecture）2.0 协议中重要的部分，它总共有 3 个通道，具有的特性包括，单个时钟边沿操作、非三态的实现方式、支持突发传输、支持分段传输以及支持多个主控制器等。

AHB 总线是 ARM 公司推出 AXI 总线之前主要推广的总线，虽然目前高性能的 SoC 中主要使用 AXI 总线，但是 AHB 总线在很多低功耗 SoC 中仍然大量使用。

12.1.3 APB

APB（Advanced Peripheral Performance Bus）是 ARM 公司提出的 AMBA（Advanced Microcontroller Bus Architecture）协议中重要的部分。APB 主要用于低带宽周边外设之间的连接，例如 UART 等。它的总线架构不像 AXI 和 AHB 那样支持多个主模块，在 APB 总线协议中里面唯一的主模块就是 APB 桥。其特性包括两个时钟周期传输，无须等待周期和回应信号，控制逻辑简单，只有 4 个控制信号。

由于 ARM 公司长时间的推广 APB 总线协议，使之几乎成为了低速设备总线的事实标准，目前很多片上低速设备和 IP 均使用 APB 接口。

12.1.4 TileLink

TileLink 总线是伯克利大学定义的一种高速片上总线协议，它诞生的初衷主要是为了定义一种标准的支持缓存一致性（Cache Coherence）的协议。并且它力图将不同的缓存一致性协议和总线的设计实现相分离，使得任何的缓存一致性协议均可遵循 TileLink 协议予以实现。

TileLink 有 5 个独立的通道，虽然 TileLink 的初衷是为了支持缓存一致性，但是它也具备片上总线的所有特性，能够支持所有存储器访问所需的操作类型。

12.1.5 总结比较

以上介绍了各种总线的优点，但各总线也有其缺点（针对蜂鸟 E200 处理器而言），总结如下。

- AXI 总线是目前应用最为广泛的高性能总线，但是主要应用于高性能的片上总线。AXI 总线有 5 个通道，分离的读和写通道能够提供很高的吞吐率，但是也需要主设备（Master）自行维护读和写的顺序，控制相对复杂，且经常在 SoC 中集成不当造成各种死锁。同时 5 个通道硬件开销过大，另外在大多数的极低功耗处理器 SoC 中都没有使用 AXI 总线。如果蜂鸟 E200 处理器核采用 AXI 总线，一方面增大硬件开销，另一方面会给用户造成负担（需要将 AXI 转换成 AHB 或者其他总线用于低功耗的 SoC），因此 AXI 总线不是特别适合蜂鸟 E200 这样的极低功耗处理器核。
- AHB 总线是目前应用最为广泛的高性能低功耗总线，ARM 的 Cortex-M 系列大多数处理器核均采用 AHB 总线。但是 AHB 总线有若干非常明显的局限性，首先其无法像 AXI 总线那样容易地添加流水线级数，其次 AHB 总线无法支持多个滞外交易（Multiple Outstanding Transaction），再次其握手协议非常别扭。将 AHB 总线转换成其他 Valid-Ready 握手类型的协议（譬如 AXI 和 TileLink 等握手总线接口）颇不容

易，跨时钟域或者整数倍时钟域更加困难，因此如果蜂鸟 E200 采用 AHB 总线作为接口也会带来同样的若干局限性。

- APB 总线是一种低速设备总线，吞吐率比较低，不适合作为主总线使用，因此更加不适用于作为蜂鸟 E200 处理器核的数据总线。
- TileLink 总线主要在伯克利大学的项目中使用，其应用并不广泛，文档也不是特别丰富，并且 TileLink 总线协议比较复杂，因此 TileLink 总线对于蜂鸟 E200 这样的极低功耗处理器核不是特别适合。

基于如上的分析，为了克服这几种缺陷，蜂鸟 E200 处理器核采用自定义的总线协议 ICB。有关 ICB 的内容详见下一节。

12.2 自定义总线协议 ICB

12.2.1 ICB 总线协议简介

为了克服上一节中所述各总线的缺陷，蜂鸟 E200 处理器开发过程中定义了一种自定义总线协议 ICB（Internal Chip Bus），用于蜂鸟 E200 处理器核内部使用，同时也可作为 SoC 中的总线使用。

ICB 总线的初衷是为了尽可能地结合 AXI 总线和 AHB 总线的优点，兼具高速性和易用性，它具有如下特性。

- 相比 AXI 和 AHB 而言，ICB 的协议控制更加简单，仅有两个独立的通道。如图 12-1 所示，读和写操作共用地址通道，共用结果返回通道。

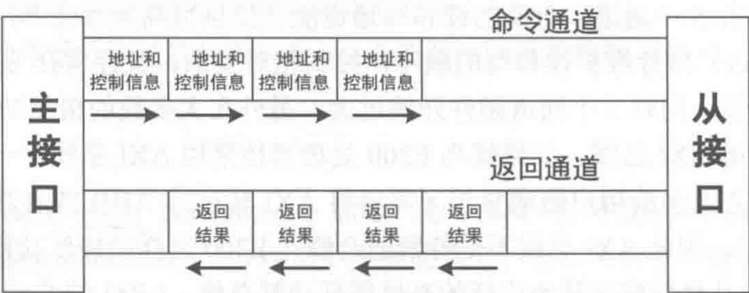


图 12-1 ICB 总线通道结构

- 与 AXI 总线一样，采用分离的地址和数据阶段。
- 与 AXI 总线一样，采用地址区间寻址，支持任意的主从数目，譬如一主一从、一主多从、多主一从、多主多从等拓扑结构。

- 与 AHB 总线一样，每个读或者写操作都会在地址通道上产生地址，而非像 AXI 中只产生起始地址。
- 与 AXI 总线一样，支持地址非对齐的数据访问，使用字节掩码（Write Mask）来控制部分写操作。
- 与 AXI 总线一样，支持多个滞外交易（Multiple Outstanding Transaction）。
- 与 AHB 总线一样，不支持乱序返回乱序完成，反馈通道必须按顺序返回结果。
- 与 AXI 总线一样，非常易于添加流水线级数以获得高频的时序。
- 协议非常简单，易于桥接转换成其他总线类型，譬如 AXI、AHB、APB 或者 TileLink 等总线。

对于蜂鸟 E200 这样的低功耗处理器而言，ICB 总线能够被用于几乎所有的场合，包括作为内部模块之间的接口、SRAM 模块接口、低速设备总线以及系统存储总线等。

12.2.2 ICB 总线协议信号

ICB 总线主要包含 2 个通道，如图 12-1 所示。

- 命令通道（Command Channel）：主要用于主设备向从设备发起读写请求。
- 返回通道（Response Channel）：主要用于从设备向主设备返回读写结果。

ICB 总线信号列表如表 12-1 所示。

表 12-1 ICB 总线信号

通　道	方向	宽度	信 号 名	介　绍
命令通道	Output	1	icb_cmd_valid	主设备向从设备发送读写请求信号
	Input	1	icb_cmd_ready	从设备向主设备返回读写接受信号
	Output	DW	icb_cmd_addr	读写地址
	Output	1	icb_cmd_read	读或是写操作的指示
	Output	DW	icb_cmd_wdata	写操作的数据
	Output	DW/8	icb_cmd_wmask	写操作的字节掩码
反馈通道	Input	1	icb_rsp_valid	从设备向主设备发送读写反馈请求信号
	Output	1	icb_rsp_ready	主设备向从设备返回读写反馈接受信号
	Input	DW	icb_rsp_rdata	读反馈的数据
	Input	1	icb_rsp_err	读或者写反馈的错误标志

12.2.3 ICB 总线协议时序

本节将描述 ICB 总线的若干典型时序。

如图 12-2 所示。

- 主设备通过 ICB 的命令通道向从设备发送写操作请求（icb_cmd_read 为低），从设备立即接收该请求（icb_cmd_ready 为高）。
- 从设备在同一个周期返回反馈且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

如图 12-3 所示。

- 主设备通过 ICB 的命令通道向从设备发送读操作请求（icb_cmd_read 为高），从设备立即接收该请求（icb_cmd_ready 为高）。
- 从设备在下一个周期返回反馈且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

clk
icb_cmd_valid
icb_cmd_ready
icb_cmd_addr
icb_cmd_read
icb_cmd_wdata
icb_cmd_wmask
icb_rsp_valid
icb_rsp_ready
icb_rsp_rdata
icb_rsp_err
addr1
dat1
msk1

图 12-2　写操作同一周期返回结果

clk
icb_cmd_valid
icb_cmd_ready
icb_cmd_addr
icb_cmd_read
icb_cmd_wdata
icb_cmd_wmask
icb_rsp_valid
icb_rsp_ready
icb_rsp_rdata
icb_rsp_err
addr
rdata

图 12-3　读操作下一周期返回结果

如图 12-4 所示。

- 主设备通过 ICB 的命令通道向从设备发送写操作请求（icb_cmd_read 为低），从设备立即接收该请求（icb_cmd_ready 为高）。
- 从设备在下一个周期返回反馈且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

如图 12-5 所示。

- 主设备向从设备通过 ICB 的命令通道发送读操作请求（icb_cmd_read 为高），从设备立即接收该请求（icb_cmd_ready 为高）。
- 从设备在 4 个周期后返回反馈且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

如图 12-6 所示。

- 主设备通过 ICB 的命令通道向从设备发送写操作请求（icb_cmd_read 为低），从设备立即接收该请求（icb_cmd_ready 为高）。
- 从设备在 4 个周期后返回反馈且结果出错（icb_rsp_err 为高），主设备立即接收该结果（icb_rsp_ready 为高）。

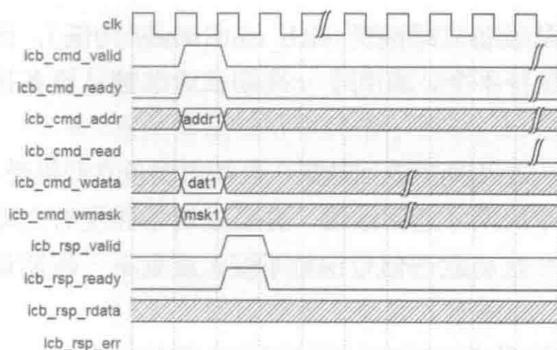


图 12-4　写操作下一周期返回结果

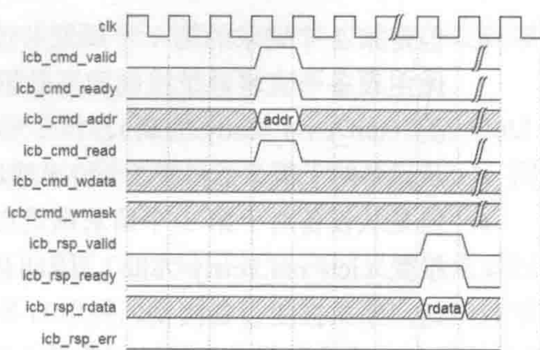


图 12-5　读操作 4 个周期返回结果

如图 12-7 所示。

- 主设备通过 ICB 的命令通道向从设备连续发送 4 个读操作请求（icb_cmd_read 为高），从设备均立即接收请求（icb_cmd_ready 为高）。
- 从设备在 4 个周期后连续返回 4 个读结果，其中前 3 个结果正确（icb_rsp_err 为低），第 4 个结果错误（icb_rsp_err 为高），主设备均立即接收此 4 个结果（icb_rsp_ready 为高）。

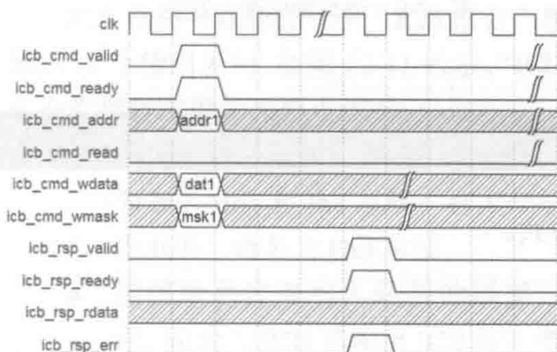


图 12-6　写操作 4 个周期返回结果

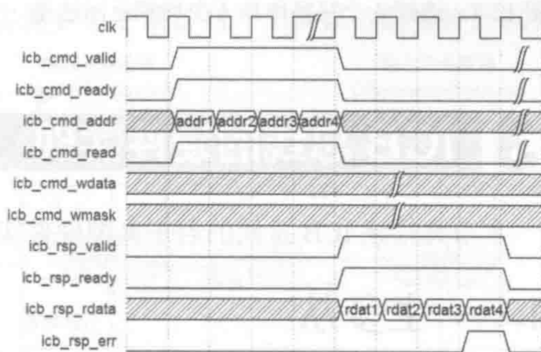


图 12-7　连续 4 个读操作均 4 个周期返回结果

如图 12-8 所示。

- 主设备通过 ICB 的命令通道向从设备连续发送 4 个写操作请求（icb_cmd_read 为低），从设备均立即接收请求（icb_cmd_ready 为高）。
- 从设备在 4 个周期后连续返回 4 个写结果，其中前 3 个结果正确（icb_rsp_err 为低），第 4 个结果错误（icb_rsp_err 为高），主设备均立即接收此 4 个结果（icb_rsp_ready 为高）。

如图 12-9 所示。

- 主设备通过 ICB 的命令通道向从设备相继发送两个读和一个写操作请求。
- 从设备立即接收了第 1 个和第 3 个请求。

- 但是第 2 个请求的第 1 个周期并没有被从设备立即接受（icb_cmd_ready 为低），因此主设备一直将地址控制和写数据信号保持不变，直到下一周期该请求被从设备接受（icb_cmd_ready 为高）。
- 从设备对于第 1 个和第 2 个请求都是在同一个周期就返回结果，且被主设备立即接受。
- 但是从设备对于第 3 个请求则是在下一个周期才返回结果，并且主设备还没有立即接受（icb_rsp_ready 为低），因此从设备一直将返回信号保持不变，直到下一周期该返回结果被主设备接受。

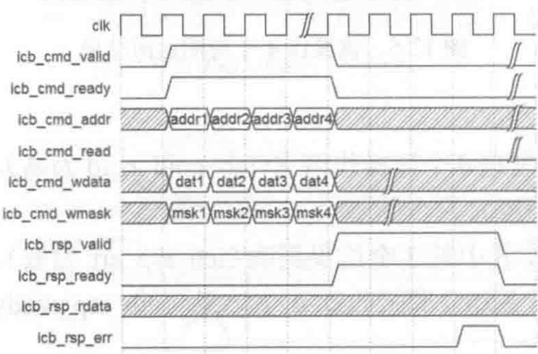


图 12-8　连续 4 个写操作均 4 个周期返回结果

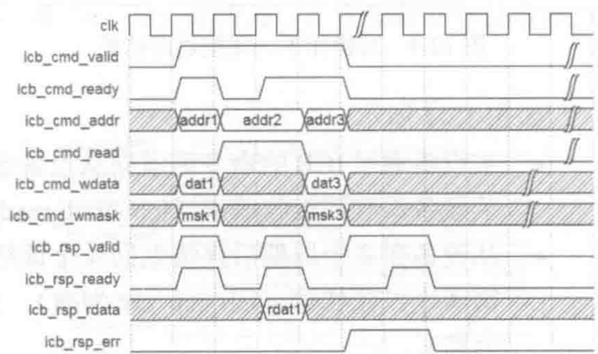


图 12-9　读写操作混合发生

12.3 ICB 总线的硬件实现

本节将描述 ICB 总线的若干典型硬件实现方式。

12.3.1 一主多从

ICB 总线可以通过一个"ICB 分发"模块实现一个主设备到多个从设备的连接。以 1 个输入 3 个输出为例，如图 12-10 所示，"ICB 分发"模块的微架构要点如下。

- 该模块有 1 个输入 ICB，命名为 In 总线；有 3 个输出 ICB，分别命名为 Out0、Out1 和 Out2 总线。
- 该模块并没有引入任何的周期延迟，即输入 ICB 和输出 ICB 在 1 个周期内穿通。

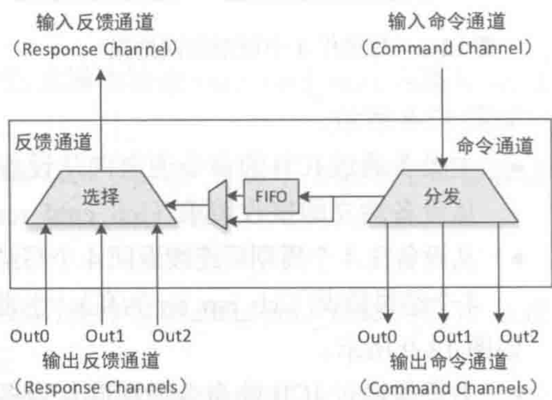


图 12-10　1 个 ICB 分发成 3 个 ICB

- 该模块的 In 总线命令通道有 1 个附属输入信号，用来指示该请求应该被分发到哪个输出 ICB 总线。该附属信号可以在顶层通过地址区间的比较判断生成所得。
- 根据附属信号中的指示信息，In 总线的命令通道被分发给 Out0、Out1 或者 Out2 输出 ICB 的命令通道。每个周期如果握手成功，则分发一个交易（Transaction），同时将“分发信息”压入 FIFO 中。
- 由于 ICB 支持多个滞外交易，Out0、Out1 或者 Out2 输出 ICB 通过反馈通道返回的结果可能需要多个周期才能返回，并且各自返回的时间点可能先后不一，因此需要被仲裁。此时可以从 FIFO 中按顺序弹出之前被压入的“分发信息”作为仲裁标准。该 FIFO 的深度决定了该模块能够支持的多个滞外交易的个数，同时由于 FIFO 先入先出的顺序性，能够保证输入 ICB 严格按照发出的顺序接收到相应的返回结果。
- 有一种极端情况，那就是当 FIFO 为空时，意味着没有滞外交易，并且当前分发的 ICB 交易可以被从设备在同一个周期内立即返回结果，那么该交易的分发信息无须被压入 FIFO，而是将其旁路使用该分发信息直接用于反馈通道选择的选通信号。

12.3.2 多主一从

ICB 总线可以通过一个“ICB 汇合”模块实现多个主设备到一个从设备的连接。以 3 个输入、1 个输出为例，如图 12-11 所示，“ICB 汇合”模块的微架构要点如下。

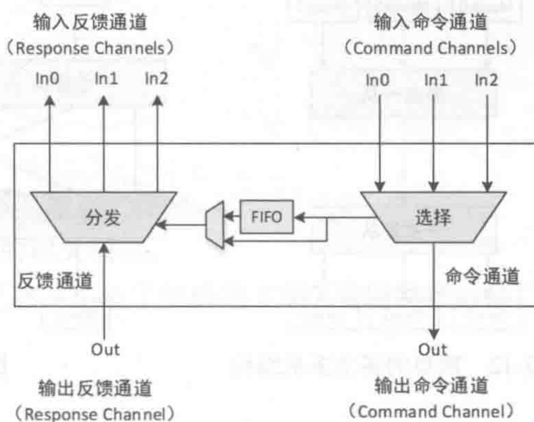


图 12-11 3 个 ICB 汇合成 1 个 ICB

- 该模块有 3 个输入 ICB，分别命名为 In0、In1 和 In2 总线；有 1 个输出 ICB，命名为 Out 总线。
- 该模块并没有引入任何的周期延迟，即输入 ICB 和输出 ICB 在 1 个周期内穿通。
- 该模块多个输入 ICB 的命令通道需要被仲裁，可以使用轮询的仲裁机制，也可以选择优先级选择的机制。以优先级选择机制为例，可以分配 In0 总线的优先级最高、In1 其次、In2 再次，通过优先级选择之后作为输出 ICB 的命令通道。每个周期如果握手成功，则仲裁发送一个交易，同时将“仲裁信息”压入 FIFO 中。
- 由于输出 ICB 通过反馈通道返回的结果一定是按顺序返回的（ICB 协议规定），因此无需担心其顺序性。但是返回的结果需要判别，并分发给对应的输入 ICB 总线，此时可以从 FIFO 中按顺序弹出之前被压入的“仲裁信息”作为分发的依据。因此该 FIFO

的深度决定了该模块能够支持的多个滞外交易的个数，同时由于 FIFO 先入先出的顺序性，能够保证各个不同的输入ICB 严格按照发出的顺序接收到相应的返回结果。

- 有一种极端情况，那就是当 FIFO 为空时，意味着没有滞外交易，并且当前仲裁的 ICB 交易可以被从设备在同一个周期内立即返回结果。那么该交易的仲裁信息无须被压入 FIFO，而是将其旁路使用该仲裁信息，直接用于反馈通道分发的选通信号。

12.3.3 多主多从

通过使用“一主多从”和“多主一从”模块的有效组合，便可以组装成为不同形式的“多主多从”模块。

第一种简单的“多主多从”模块如图 12-12 所示。通过将“多主一从”和“一主多从”模块,直接对接，便可达到多主多从的效果。但是其缺陷是所有的主 ICB 总线均需要通过中间一条公用的 ICB 总线，吞吐率受限。

第二种稍微复杂的“多主多从”模块如图 12-13 所示。通过使用多个“一主多从”和“多主一从”模块交织组装成为“多主多从”的交叉开关（Crossbar）结构。该结构使得每个主接口和从接口之间均有专用的通道，但是其缺陷是面积开销很大，并且设计不当容易造成死锁。

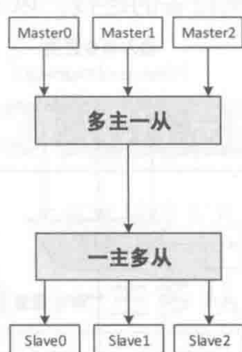


图 12-12 简单的多主多从结构

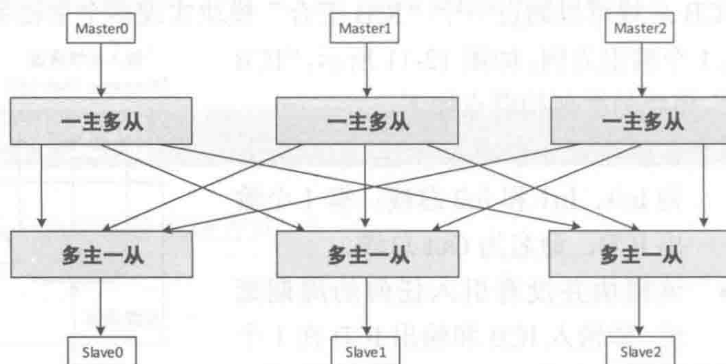


图 12-13 交叉开关的多主多从结构

12.4 蜂鸟 E200 处理器核 BIU

本节将介绍蜂鸟 E200 处理器核 BIU 模块的微架构及其源码分析。

12.4.1 BIU 简介

BIU 在蜂鸟 E200 处理器核中的位置如图 12-14 所示。

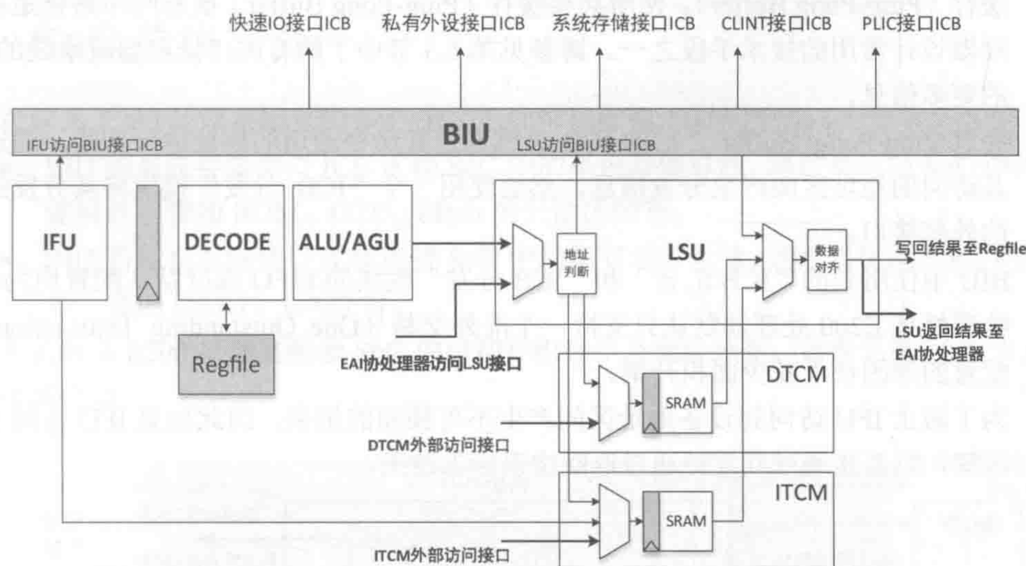


图 12-14　BIU 在蜂鸟 E200 处理器核中的位置

BIU 主要负责接受来自 IFU（Instruction Fetch Unit）和 LSU（Load Store Unit）单元的存储器访问请求，并且使用标准的 ICB 接口。然后通过判断其访问的地址区间来访问外部的不同接口，包括以下 5 种。

- 快速 IO 接口。
- 私有外设接口。
- 系统存储接口。
- CLINT 接口，参见第 13.5.5 节了解 CLINT 的相关信息。
- PLIC 接口，参见第 13.5.6 节了解 PLIC 的相关信息。

有关这些接口和组件的信息和作用，请参见第 4.2 节了解蜂鸟 E200 处理器核的接口信息，和第 18.2 节了解 SoC 结构的更多信息。

12.4.2 BIU 微架构

BIU 的微架构如图 12-15 所示，其要点如下。

- BIU 有两组输入 ICB 总线接口，分别来自于 IFU 和 LSU 单元。请参见第 7.3 节了解 IFU 的更多信息，请参见第 11.4 节了解 LSU 的更多信息。
- 两组输入 ICB 总线经过一个“ICB 汇合”模块，将其汇合成为一组 ICB 总线，采用的仲裁机制是优先级仲裁，LSU 总线具有更高的优先级。
- 为了砍断外界与处理器核内部之间的时序路径，在汇合的 ICB 总线处插入一组乒乓

缓存（Ping-Pong Buffer）。使用乒乓缓存（Ping-Pong Buffer）砍断时序路径是高速处理器设计常用的技术手段之一。请参见第 6.3 节中了解有关“处理器流水线的反压”的更多信息。

- 经过 Ping-Pong Buffer 之后的 ICB 总线通过其命令通道的地址进行判断，通过判断其访问的地址区间产生分发信息，然后使用一个“ICB 分发”模块将其分发给不同的外部接口。
- BIU 中使用到的“ICB 汇合”和“ICB 分发”模块的 FIFO 深度默认配置均为 1，意味着蜂鸟 E200 处理器默认只支持一个滞外交易（One Outstanding Transaction），此配置的原因在于减少面积开销。
- 为了防止 IFU 访问到设备地址区间产生不可预知的结果，因此如果 IFU 访问了设备区间，则直接通过其反馈通道返回错误标志结果。

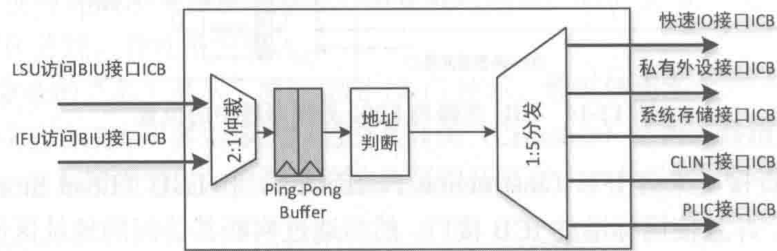


图 12-15 BIU 微架构图

12.4.3 BIU 源码分析

BIU 的相关源代码在 e200_opensource 目录的结构如下。

```
e200_opensource
    |----rtl                          // 存放 RTL 的目录
         |----e203                    // E203 核和 SoC 的 RTL 目录
              |----general            // 存放一些公用的通用 RTL 代码
              |----core               // 存放 e203 Core 的 RTL 代码
                   |----e203_biu.v    // BIU 的源代码
```

由于 BIU 的源代码比较简单，请读者到 GitHub 的 e200_opensource 项目中自行阅读。

12.5 蜂鸟 E200 处理器 SoC 总线

由于蜂鸟 E200 BIU 模块直接对接外部 SoC 的总线，因此本节将顺便介绍蜂鸟 E200 处理器的 SoC 总线，包括其微架构和源码分析。

12.5.1 SoC 总线简介

在蜂鸟 E200 处理器配套的 SoC 中，总线结构如图 12-16 所示。

- BIU 的系统存储接口 ICB 连接 SoC 中的系统存储总线，通过其访问 SoC 中的若干存储组件，譬如 ROM、OTP、Flash 的只读区间等。
- BIU 的私有外设接口 ICB 连接 SoC 中的私有设备总线，通过其访问 SoC 中的若干设备，譬如 UART、GPIO 等。

有关蜂鸟 E200 处理器配套 SoC 的结构和组件以及地址分配，请参见第 18.2 节了解更多 SoC 详细信息。

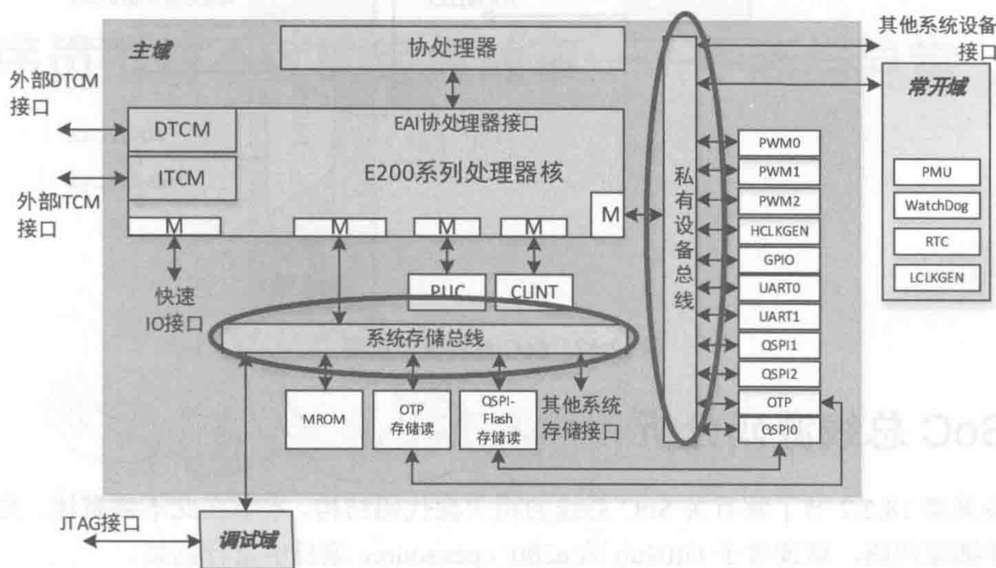


图 12-16 蜂鸟 E200 处理器 SoC 总线示意图

12.5.2 SoC 总线微架构

SoC 总线的微架构如图 12-17 所示。其要点如下。

- 私有外设接口 ICB 总线通过其命令通道的地址进行判断，通过其访问的地址区间产生分发信息，然后使用一个“ICB 分发”模块将其分发给不同的外设 ICB 接口。
- 系统存储接口 ICB 总线通过其命令通道的地址进行判断，通过其访问的地址区间产生分发信息，然后使用一个“ICB 分发”模块将其分发给不同的存储模块 ICB 接口。
- 与 BIU 微架构同理，如果任何 ICB 路径中存在着时序的关键路径，可以插入一组乒乓缓存砍断前后的时序路径。同理，如果任何 ICB 路径需要跨越异步时钟域或者整

数倍分频时钟域，也可以插入相应的异步 FIFO 或者流水线级数。

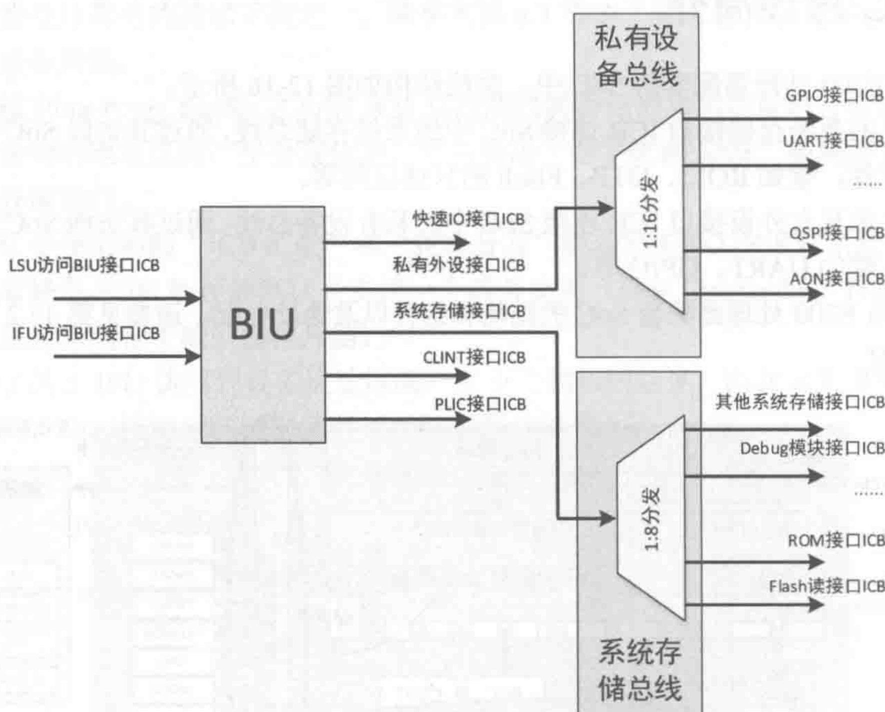


图 12-17　SoC 总线微架构图

12.5.3　SoC 总线源码分析

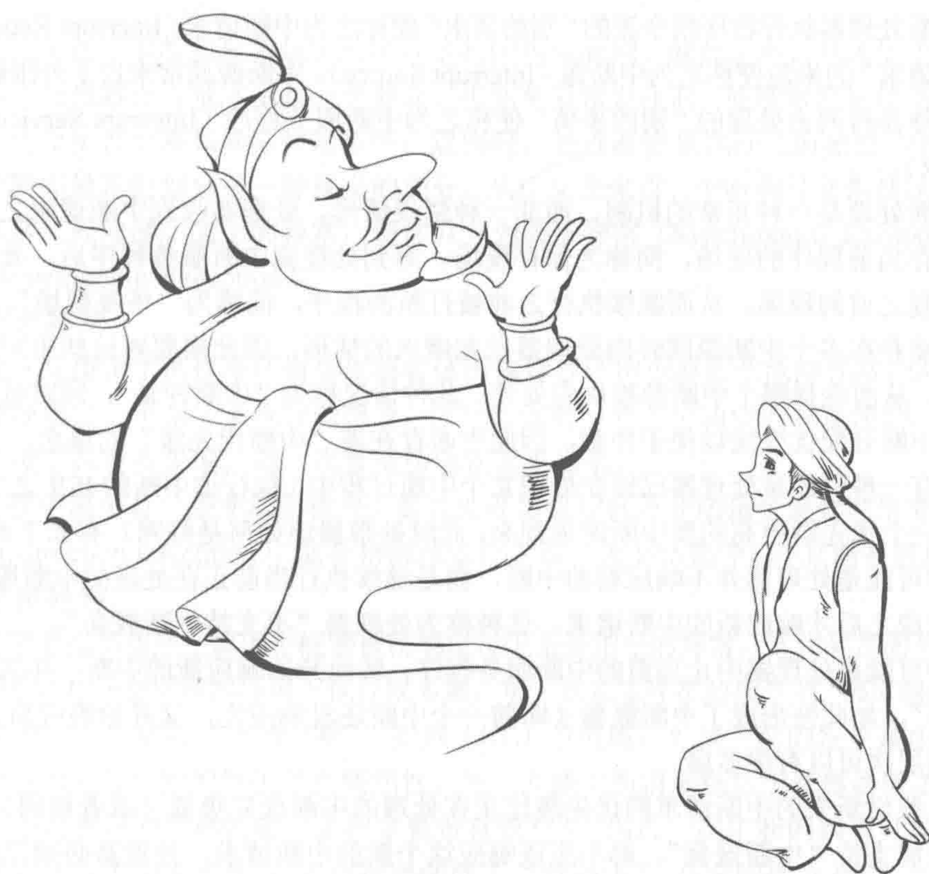
请参见第 18.2.2 节了解有关 SoC 总线的相关源代码结构，本节在此不做赘述。关于 SoC 总线的详细源代码，请读者于 GitHub 的 e200_opensource 项目中自行阅读。

12.6 总结

自定义 ICB 总线是蜂鸟 E200 处理器核和配套 SoC 的一个特点，通过统一而简单的 ICB 总线，蜂鸟 E200 处理器核和配套 SoC 能够在最大程度上实现灵活性和自主性。由于 ICB 总线极其简单，能够非常容易地被转换成为其他的任何一种流行总线，譬如 AHB、AXI、Wishbone 总线等，使得蜂鸟 E200 处理器核能够取得尽量多的普适性。此外，ICB 总线非常便于插入流水线级数，因此蜂鸟 E200 处理器核和 SoC 总线均能够运行到相当高的主频。

第13章　不得不说的故事——中断和异常

程序员不得不读的经典故事——中断与异常



中断和异常虽说本身不是一种指令，但却是处理器指令集架构中非常重要的一环，任何一种指令集架构都会安排专门的章节定义和详解其中断和异常的行为，可以说中断和异常是不得不说的故事。

本章将介绍 RISC-V 架构定义的中断和异常机制，以及蜂鸟 E200 处理器核中断和异常的硬件微架构和源码分析。

13.1 中断和异常概述

13.1.1 中断概述

中断（Interrupt）机制，即处理器核在顺序执行程序指令流的过程中突然被别的请求打断而中止执行当前的程序，转而去处理别的事情，待其处理完了别的事情，然后重新回到之前程序中断的点继续执行之前的程序指令流，其要点如下。

- 打断处理器执行程序指令流的“别的请求”便称之为中断请求（Interrupt Request），“别的请求”的来源便称之为中断源（Interrupt Source）。中断源通常来自于外围硬件设备。
- 处理器转而去处理的“别的事情”便称之为中断服务程序（Interrupt Service Routine，ISR）。
- 中断处理是一种正常的机制，而非一种错误情形。处理器收到中断请求之后，需要保存当前程序的现场，简称为保存现场。等到处理完中断服务程序后，处理器需要恢复之前的现场，从而继续执行之前被打断的程序，简称为“恢复现场”。
- 可能存在多个中断源同时向处理器发起请求的情形，因此需要对这些中断源进行仲裁，从而选择哪个中断源被优先处理。此种情况称为“中断仲裁”，同时可以给不同的中断分配优先级以便于仲裁，因此中断存在着“中断优先级”的概念。
- 还有一种可能是处理器已经在处理某个中断过程中（执行该中断的 ISR 之中），此时有一个优先级更高的新中断请求到来，此时处理器该如何是好呢？有如下两种可能。

第一种可能是处理器并不响应新的中断，而是继续执行当前正在处理的中断服务程序，待到彻底完成之后才响应新的中断请求，这种称为处理器“不支持中断嵌套”。

第二种可能是处理器中止当前的中断服务程序，转而开始响应新的中断，并执行其“中断服务程序”，如此便形成了中断嵌套（即前一个中断还没响应完，又开始响应新的中断），并且嵌套的层次可以有很多层。

注意：假设新来的中断请求的优先级比正在处理的中断优先级低（或者相同），则不管处理器是否能支持“中断嵌套”，都不应该响应这个新的中断请求，处理器必须完成当前的

中断服务程序之后才考虑响应新的中断请求（因为新中断请求的优先级并不比当前正在处理的中断优先级高）。

13.1.2 异常概述

异常（Exception）机制，即处理器核在顺序执行程序指令流的过程中突然遇到了异常的事情而中止执行当前的程序，转而去处理该异常，其要点如下。

- 处理器遇到的“异常的事情”称为异常（Exception）。异常与中断的最大区别在于中断往往是一种外因，而异常是由处理器内部事件或程序执行中的事件引起的，譬如本身硬件故障、程序故障，或者执行特殊的系统服务指令而引起的，简而言之是一种内因。
- 与中断服务程序类似，处理器也会进入异常服务处理程序。
- 与中断类似，可能存在多个异常同时发生的情形，因此异常也有优先级，并且也可能发生多重异常的嵌套。

13.1.3 广义上的异常

如上一节所述，中断和异常最大的区别是起因内外有别。除此之外，从本质上来讲，中断和异常对于处理器而言基本上是一个概念。中断和异常发生时，处理器将暂停当前正在执行的程序，转而执行中断和异常处理程序；返回时，处理器恢复执行之前被暂停的程序。

因此中断和异常的划分是一种狭义的划分。从广义上来讲，中断和异常都被认为是一种广义上的异常。处理器广义上的异常，通常只分为同步异常（Synchronous Exception）和异步异常（Asynchronous Exception）。

1．同步异常

同步异常是指由于执行程序指令流或者试图执行程序指令流而造成的异常。这种异常的原因能够被精确定位于某一条执行的指令。同步异常的另外一个通俗的表象便是，无论程序在同样的环境下执行多少遍，每一次都能精确地重现出来。

譬如，程序流中有一条非法的指令，那么处理器执行到该非法指令便会产生非法指令异常（Illegal Instruction Exception），能被精确地定位于这一条非法指令，并且能够被反复重现。

2．异步异常

异步异常是指那些产生原因不能够被精确定位于某条指令的异常。异步异常的另外一个通俗的表象便是，程序在同样的环境下执行很多遍，每一次发生异常的指令 PC 都可能会不一样。

最常见的异步异常是“外部中断”。如第 13.1.1 节所述，外部中断的发生是由外围设备驱动的，一方面外部中断的发生带有偶然性，另一方面中断请求抵达于处理器核之时，处理器的程序指令流执行到具体的哪一条指令更带有偶然性。因此一次中断的到来可能会巧遇到某一条“正在执行的不幸指令”，而该指令便成了“背锅侠”。在它的指令 PC 所在之处，程

序便停止执行，并转而响应中断去执行中断服务程序。但是当程序重复执行时，却很难会出现同一条指令反复“背锅”的精确情形。

对于异步异常，根据其响应异常后的处理器状态，又可以分为两种。

（1）精确异步异常（Precise Asynchronous Exception）：指响应异常后的处理器状态能够精确反映为某一条指令的边界，即某一条指令执行完之后的处理器状态。

（2）非精确异步异常（Imprecise Asynchronous Exception）：指响应异常后的处理器状态无法精确反映为某一条指令的边界，即可能是某一条指令执行了一半然后被打断的结果，或者是其他模糊的状态。

常见的典型同步异常和异步异常如表 13-1 所示，此表可以帮助读者更加理解同步异常和异步异常的区别。

表 13-1 同步异常和异步异常

	典型异常
同步异常	• 取指令访问到非法的地址区间 譬如外设模块的地址区间往往是不可能存放指令代码的，因此其属性是“不可执行”，并且还是读敏感的（Read Sensitive）。如果某条指令的 PC 位于外设区间，则会造成取指令错误。这种错误能够精确的定位到是哪一条指令 PC 造成的
	• 读写数据访问地址属性出错 譬如有的地址区间的属性是只读或者只写的，假设 Load 或者 Store 指令以错误的方式访问了地址区间（譬如写了只读的区间），这种错误方式能够被存储器保护单元（Memory Protection Unit，MPU）或者存储器管理单元（Memory Management Unit，MMU）及时探测出来，则能够精确地定位到是哪一条 Load 或 Store 指令访存造成的 MPU 和 MMU 是分别对地址进行保护和管理的硬件单元，本书限于篇幅在此对其不做赘述，感兴趣的读者请自行查阅其他资料
	• 取指令地址非对齐错误 处理器指令集架构往往规定指令存放在存储器中的地址必须是对齐的，譬如 16 位长的指令往往要求其 PC 值必须是 16 位对齐的。假设该指令的 PC 值不对齐，则会造成取指令不对齐错误。这种错误能够精确的定位到是哪一条指令 PC 造成的
	• 非法指令错误 处理器如果对指令进行译码发现这是一条非法的指令（譬如不存在的指令编码），则会造成非法指令错误。这种错误能够精确的定位到是哪一条指令造成的
	• 执行调试断点指令 处理器指令集架构往往会定义若干条调试指令，譬如断点（EBREAK）指令。当执行到该指令时处理器便会发生异常进入异常服务程序。该指令往往用于调试器（Debugger）使用，譬如设置断点。这种异常能够被精确地定位于具体是哪一条 EBREAK 指令造成的
精确异步异常	• 外部中断 外部中断是最常见的精确异步异常，第 13.1.1 节已对其专门论述，此处不再赘述

续表

	典型异常
非精确异步异常	• 读写存储器出错 读写存储器出错是另外一种最常见的非精确异步异常，由于访问存储器（简称访存）需要一定的时间，处理器往往不可能坐等该访问结束（否则性能会很差），而是会继续执行后续的指令。等到访存结果从目标存储器返回来之后，发现出现了访存错误并汇报异常，但是处理器此时可能已经执行到了后续的某条指令，难以精确定位。并且存储器返回的时间延迟也具有偶然性，无法被精确地重现 这种异步异常的另外一个常见示例便是写操作将数据写入缓存行（Cache Line）中，然后该缓存行经过很久才被替换出来，写回外部存储器，但是写回外部存储器返回结果出错。此时处理器可能已经执行过了后续成百上千条指令，到底是哪一条指令当时写的这个地址的缓存行早已是“前朝旧事”，不可能被精确定位，更不要说复现了。有关缓存的细节，本书限于篇幅在此对其不做赘述，感兴趣的读者请自行查阅其他资料

13.2 RISC-V 架构异常处理机制

本节将介绍 RISC-V 架构的异常处理机制。如附录 A1 中所述，当前 RISC-V 架构文档主要分为“指令集文档”和“特权架构文档”。RISC-V 架构的异常处理机制定义在“特权架构文档”中。请参见第 2.1.1 节了解 RISC-V 架构文档的下载地址。

如第 13.1.3 节中所述，狭义的中断和异常均可以被归于广义的异常范畴，因此本书自此将用“异常”作为统一概念进行论述，其包含了狭义上的“中断”和“异常”。

RISC-V 的架构不仅可以有机器模式（Machine Mode）的工作模式，还可以有用户模式（User Mode）、监督模式（Supervisor Mode）等工作模式。在不同的模式下均可以产生异常，并且有的模式也可以响应中断。请参见附录 A8 了解更多工作模式的信息。

RISC-V 架构要求机器模式是必须具备的模式，其他的模式均是可选而非必选的模式。因此为了简化模型便于读者理解，且由于蜂鸟 E200 只实现了机器模式，本章仅介绍基于机器模式的异常处理机制。

13.2.1 进入异常

进入异常时，RISC-V 架构规定的硬件行为可以简述如下。

（1）停止执行当前程序流，转而从 CSR 寄存器 mtvec 定义的 PC 地址开始执行。

（2）进入异常不仅会让处理器跳转到上述的 PC 地址开始执行，还会让硬件同时更新其他几个 CSR 寄存器，分别是以下 4 个寄存器。

- 机器模式异常原因寄存器 mcause（Machine Cause Register）
- 机器模式异常 PC 寄存器 mepc（Machine Exception Program Counter）

- 机器模式异常值寄存器 mtval（Machine Trap Value Register ）
- 机器模式状态寄存器 mstatus（Machine Status Register）

下文将分别予以详述。

1. 从 mtvec 定义的 PC 地址开始执行

RISC-V 架构规定，在处理器的程序执行过程中，一旦遇到异常发生，则终止当前的程序流，处理器被强行跳转到一个新的 PC 地址。该过程在 RISC-V 的架构中定义为“陷阱（trap）”，字面含义为“跳入陷阱”，更加准确的意译为“进入异常”。

RISC-V 处理器 trap 后跳入的 PC 地址由一个叫做机器模式异常入口基地址寄存器 mtvec（Machine Trap-Vector Base-Address Register）的 CSR 寄存器指定，其要点如下。

（1）mtvec 寄存器是一个可读可写的 CSR 寄存器，因此软件可以编程更改其中的值。

（2）mtvec 寄存器的详细格式如图 13-1 所示，其中的最低 2 位是 MODE 域，高 30 位是 BASE 域。



图 13-1 mtvec 寄存器的格式

- 假设 MODE 的值为 0，则所有的异常响应时处理器均跳转到 BASE 值指示的 PC 地址。
- 假设 MODE 的值为 1，则狭义的异常发生时，处理器跳转到 BASE 值指示的 PC 地址；狭义的中断发生时，处理器跳转到 BASE+4*CAUSE 值指示的 PC 地址。CAUSE 的值表示中断对应的异常编号（Exception Code），如图 13-3 所示。譬如机器计时器中断（Machine Timer Interrupt）的异常编号为 7，则其跳转的地址为 BASE+4×7=BASE+28= BASE+0x1c。

2. 更新 CSR 寄存器 mcause

RISC-V 架构规定，在进入异常时，机器模式异常原因寄存器 mcause（Machine Cause Register）被同时更新，以反映当前的异常种类，软件可以通过读此寄存器查询造成异常的具体原因。

mcause 寄存器的详细格式如图 13-2 所示，其中最高 1 位为 Interrupt 域，低 31 位为异常编号域。此两个域的组合表示值如图 13-3 所示，用于指示 RISC-V 架构定义的 12 种中断类型和 16 种异常类型。

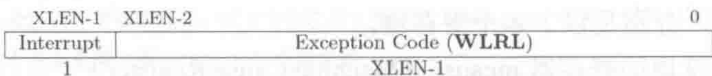


图 13-2 mcause 寄存器的格式

3. 更新 CSR 寄存器 mepc

RISC-V 架构定义异常的返回地址由机器模式异常 PC 寄存器 mepc（Machine Exception Program Counter）保存。在进入异常时，硬件将自动更新 mepc 寄存器的值为当前遇到异常的指令 PC 值(即当前程序的停止执行点)。该寄存器将作为异常的返回地址，在异常结束之后，能够使用它保存的 PC 值回到之前被停止执行的程序点。

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	⩾12	*Reserved*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	⩾16	*Reserved*

图 13-3 mcause 寄存器中的 Exception Code

（1）值得注意的是，虽然 mepc 寄存器会在异常发生时自动被硬件更新，但是 mepc 寄存器本身也是一个可读可写的寄存器，因此软件也可以直接写该寄存器以修改其值。

（2）对于狭义的中断和狭义的异常而言，RISC-V 架构定义其返回地址（更新的 mepc 值）有些细微差别。

- 出现中断时，中断返回地址 mepc 的值被更新为下一条尚未执行的指令。
- 出现异常时，中断返回地址 mepc 的值被更新为当前发生异常的指令 PC。注意：如果异常由 ecall 或 ebreak 产生，由于 mepc 的值被更新为 ecall 或 ebreak 指令自己的 PC。因此在异常返回时，如果直接使用 mepc 保存的 PC 值作为返回地址，则会再次跳回 ecall 或者 ebreak 指令，从而造成死循环（执行 ecall 或者 ebreak 指令导致重新进入异常）。正确的做法是在异常处理程序中软件改变 mepc 指向下一条指令，由于现在 ecall/ebreak（或 c.ebreak）是 4（或 2）字节指令，因此改写设定 mepc=mepc+4（或+2）即可。

4. 更新 CSR 寄存器 mtval

RISC-V 架构规定，在进入异常时，硬件将自动更新机器模式异常值寄存器 mtval（Machine Trap Value Register ），以反映引起当前异常的存储器访问地址或者指令编码。

- 如果是由存储器访问造成的异常，譬如遭遇硬件断点、取指令、存储器读写造成的异常，则将存储器访问的地址更新到 mtval 寄存器中。
- 如果是由非法指令造成的异常，则将该指令的指令编码更新到 mtval 寄存器中。

注意：mtval 寄存器又名 mbadaddr 寄存器，在某些早期版本的 RISC-V 编译器中仅识别 mbadaddr 名称。

5. 更新 CSR 寄存器 mstatus

RISC-V 架构规定，在进入异常时，硬件将自动更新机器模式状态寄存器 mstatus（Machine

Status Register）的某些域。

（1）mstatus寄存器的详细格式如图13-4所示，其中的MIE域表示在Machine Mode下中断全局使能。

31	30 ... 23	22	21	20	19	18	17
SD	WPRI	TSR	TW	TVM	MXR	SUM	MPRV
1	8	1	1	1	1	1	1

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS[1:0]	FS[1:0]	MPP[1:0]	WPRI	SPP	MPIE	WPRI	SPIE	UPIE	MIE	WPRI	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

图13-4　mstatus寄存器的格式

- 当该MIE域的值为1时，表示Machine Mode下所有中断的全局打开。
- 当该MIE域的值为0时，表示Machine Mode下所有中断的全局关闭。

（2）RISC-V架构规定，异常发生时有如下情况。

- MPIE域的值被更新为异常发生前MIE域的值。MPIE域的作用是在异常结束之后，能够使用MPIE的值恢复出异常发生之前的MIE值。
- MIE的值则被更新成为0（意味着进入异常服务程序后中断被全局关闭，所有的中断都将被屏蔽不响应）。
- MPP的值被更新为异常发生前的模式。MPP域的作用是在异常结束之后，能够使用MPP的值恢复出异常发生之前的工作模式。对于只支持机器模式（Machine Mode Only）的处理器核（譬如蜂鸟E200），则MPP的值永远为二进制值11。

注意：由于本书为简化知识模型，在此仅介绍“只支持机器模式”的架构，因此对SIE、UIE、SPP、SPIE等不做赘述。对其感兴趣的读者请参见RISC-V“特权架构文档”原文。

13.2.2 退出异常

当程序完成异常处理之后，最终需要从异常服务程序中退出，并返回主程序。RISC-V架构定义了一组专门的退出异常指令（Trap-Return Instructions），包括MRET、SRET、和URET。其中MRET指令是必备的，而SRET和URET指令仅在支持监督模式和用户模式的处理器中使用。

注意：由于本书为简化知识模型，在此仅介绍“只支持机器模式”的架构，对SRET和URET指令不做赘述。对其感兴趣的读者请参见RISC-V“特权架构文档”原文。

在机器模式下退出异常时，软件必须使用MRET指令。RISC-V架构规定，处理器执行MRET指令后的硬件行为如下。

- 停止执行当前程序流，转而从CSR寄存器mepc定义的PC地址开始执行。

- 执行 MRET 指令不仅会让处理器跳转到上述的 PC 地址开始执行，还会让硬件同时更新 CSR 寄存器机器模式状态寄存器 mstatus（Machine Status Register）。

下文将分别予以详述。

1. 从 mepc 定义的 PC 地址开始执行

在第 13.2.1 节中曾经提及，在进入异常时，mepc 寄存器被同时更新，以反映当时遇到异常的指令的 PC 值。通过这个机制，意味着 MRET 指令执行后处理器回到了当时遇到异常的指令的 PC 地址，从而可以继续执行之前被中止的程序流。

2. 更新 CSR 寄存器 mstatus

mstatus 寄存器的详细格式如图 13-4 所示。RISC-V 架构规定，在执行 MRET 指令后，硬件将自动更新机器模式状态寄存器 mstatus（Machine Status Register）的某些域。

RISC-V 架构规定，执行 MRET 指令退出异常时有如下情况。

- mstatus 寄存器 MIE 域的值被更新为当前 MPIE 的值。
- mstatus 寄存器 MPIE 域的值则被更新为 1。

在第 13.2.1 节中曾提及，在进入异常时，MPIE 的值曾经被更新为异常发生前的 MIE 值。而 MRET 指令执行后，再次将 MIE 域的值更新为 MPIE 的值。通过这个机制，则意味着 MRET 指令执行后，处理器的 MIE 值被恢复成异常发生之前的值（假设之前的 MIE 值为 1，则意味着中断被重新全局打开）。

13.2.3 异常服务程序

如第 13.2.1 节中所述，当处理器进入异常后，即开始从 mtvec 寄存器定义的 PC 地址执行新的程序。该程序通常为异常服务程序，并且程序还可以通过查询 mcause 中的异常编号（Exception Code）决定进一步跳转到更具体的异常服务程序。譬如当程序查询 mcause 中的值为 0x2，则得知该异常是非法指令错误（Illegal Instructions）引起的，因此可以进一步跳转到非法指令错误异常服务子程序中去。

图 13-5 所示为一异常入口程序实例片段，程序通过读取 mcause 的值，进而判断异常的类型，从而进入不同的异常服务子程序。

```
uintptr_t handle_trap(uintptr_t mcause, uintptr_t epc)
{
  if (0){
    // External Machine-Level interrupt from PLIC
  } else if ((mcause & MCAUSE_INT) && ((mcause & MCAUSE_CAUSE) == IRQ_M_EXT)) {
    handle_m_ext_interrupt();
    // External Machine-Level interrupt from PLIC
  } else if ((mcause & MCAUSE_INT) && ((mcause & MCAUSE_CAUSE) == IRQ_M_TIMER)){
    handle_m_time_interrupt();
  }
  else {
    write(1, "trap\n", 5);
    _exit(1 + mcause);
  }
  return epc;
}
```

图 13-5 异常服务程序示例片段

注意：由于RISC-V架构规定的进入异常和退出异常机制中没有硬件自动保存和恢复上下文的操作，因此需要软件明确地使用指令进行上下文的保存和恢复。

13.3 RISC-V架构中断定义

13.3.1 中断类型

RISC-V架构定义的中断类型分为4种。

- 外部中断（External Interrupt）
- 计时器中断（Timer Interrupt）
- 软件中断（Software Interrupt）
- 调试中断（Debug Interrupt）

下文将分别予以详述。

1. 外部中断

RISC-V架构定义的外部中断要点如下。

（1）外部中断是指来自于处理器核外部的中断，譬如外部设备UART、GPIO等产生的中断。

（2）RISC-V架构在机器模式、监督模式和用户模式下均有对应的外部中断。由于本书为简化知识模型，在此仅介绍“只支持机器模式”的架构，因此仅介绍机器模式外部中断。

（3）机器模式外部中断（Machine External Interrupt）的屏蔽由CSR寄存器mie中的MEIE域控制，等待（Pending）标志则反映在CSR寄存器mip中的MEIP域。请参见第13.3.2节和第13.3.3节了解mip和mip寄存器的更多详情。

（4）机器模式外部中断可以作为处理器核的一个单比特输入信号，假设处理器需要支持很多个外部中断源，RISC-V架构定义了一个平台级别中断控制器（Platform Level Interrupt Controller，PLIC）可用于多个外部中断源的优先级仲裁和派发。

- PLIC可以将多个外部中断源仲裁为一个单比特的中断信号送入处理器核，处理器核收到中断进入异常服务程序后，可以通过读PLIC的相关寄存器查看中断源的编号和信息。
- 处理器核在处理完相应的中断服务程序后，可以通过写PLIC的相关寄存器和具体的外部中断源的寄存器，从而清除中断源（假设中断来源为GPIO，则可通过GPIO模块的中断相关寄存器清除该中断）。
- 有关PLIC的详情请参见附录C。

（5）虽然RISC-V架构只明确定义了一个机器模式外部中断，同时明确定义可通过PLIC在外部管理众多的外部中断源将其仲裁成为一根机器模式外部中断信号传递给处理器核。但是RISC-V架构也预留了大量的空间供用户扩展其他外部中断类型，如以下3种。

- CSR 寄存器 mie 和 mip 的高 20 位可以用于扩展控制其他的自定义中断类型。请参见第 13.3.2 节和第 13.3.3 节了解 mie 和 mip 寄存器的更多详情。
- 用户甚至可以自定义若干组新的 mie<n>和 mip<n>寄存器以支持更多自定义中断类型。
- CSR 寄存器 mcause 的中断异常编号域为 12 及以上的值，均可以用于其他自定义中断的异常编号（Exception Code）。因此理论上通过扩展，RISC-V 架构可以支持无数根自定义的外部中断（External Interrupt）信号直接输入给处理器核。请参见第 13.2.1 节了解 mcause 寄存器的更多详情。

2．计时器中断

RISC-V 架构定义的计时器中断要点如下。

（1）计时器中断是指来自计时器的中断。

（2）RISC-V 架构在机器模式、监督模式和用户模式下均有对应的计时器中断。由于本书为简化知识模型，在此仅介绍“只支持机器模式”的架构，因此仅介绍机器模式计时器中断（Machine Timer Interrupt）。

（3）机器模式计时器中断的屏蔽由 mie 寄存器中的 MTIE 域控制，等待（Pending）标志则反映在 mip 寄存器中的 MTIP 域。请参见第 13.3.2 节和第 13.3.3 节了解 mip 和 mip 寄存器的更多详情。

（4）RISC-V 架构定义了系统平台中必须有一个计时器，并给该计时器定义了两个 64 位宽的寄存器 mtime（如图 13-6 所示）和 mtimecmp（如图 13-7 所示）。mtime 寄存器用于反映当前计时器的计数值，mtimecmp 用于设置计时器的比较值。当 mtime 中的计数值大于或者等于 mtimecmp 中设置的比较值时，计时器便会产生计时器中断。计时器中断会一直拉高，直到软件重新写 mtimecmp 寄存器的值，使得其比较值大于 mtime 中的值，从而将计时器中断清除。

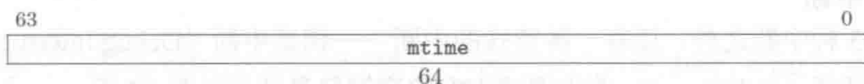


图 13-6　mtime 寄存器

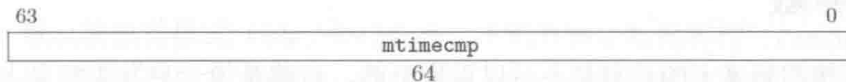


图 13-7　mtimecmp 寄存器

- 值得注意的是，RISC-V 架构并没有定义 mtime 寄存器和 mtimecmp 寄存器为 CSR 寄存器，而是定义其为存储器地址映射（Memory Address Mapped）的系统寄存器，具体的存储器映射（Memory Mapped）地址 RISC-V 架构并没有规定，而是交由 SoC 系统集成者实现。

蜂鸟 E200 配套 SoC 中的 mtime 和 mtimecmp 寄存器的实现及存储器地址映射，请参见第 13.5.5 节中有关 CLINT 模块的介绍。

- 另一点值得注意的是，RISC-V 架构定义 mtime 定时器为实时（Real-Time）计时器，系统必须以一种恒定的频率作为计时器的时钟。该恒定的时钟频率必须为低速的电源常开的（Always-on）时钟，低速是为了省电，常开是为了提供准确的计时。

3．软件中断

RISC-V 架构定义的软件中断要点如下。

（1）软件中断是指来自软件自己触发的中断。

（2）由于 RISC-V 架构在机器模式、监督模式和用户模式下均有对应的软件中断。由于本书为简化知识模型，在此仅介绍“只支持机器模式”的架构，因此仅介绍机器模式软件中断（Machine Software Interrupt）。

（3）机器模式软件中断的屏蔽由 mie 寄存器中的 MSIE 域控制，等待（Pending）标志则反映在 mip 寄存器中的 MSIP 域。

（4）RISC-V 架构定义的机器模式软件中断可以通过软件写 1 至 msip 寄存器来触发。

- 注意：此 msip 寄存器和 mip 寄存器中的 MSIP 域命名不可混淆。且 RISC-V 架构并没有定义 msip 寄存器为 CSR 寄存器，而是定义其为存储器地址映射的系统寄存器，具体的存储器映射地址 RISC-V 架构并没有规定，而是交由 SoC 系统集成者实现。
- 蜂鸟 E200 配套 SoC 中的 msip 寄存器的实现及存储器地址映射，请参见第 13.5.5 节中有关 CLINT 模块的介绍。

（5）当软件写 1 至 msip 寄存器触发了软件中断之后，CSR 寄存器 mip 中的 MSIP 域便会置高，反映其等待状态。软件可通过写 0 至 msip 寄存器来清除该软件中断。

4．调试中断

除了上述 3 种中断之外，还有一种特殊的中断——调试中断（Debug Interrupt）。此中断专用于实现调试器（Debugger），关于调试方案的详细信息请参见第 14 章。

13.3.2 中断屏蔽

RISC-V 架构的狭义上的异常是不可以被屏蔽的，也就是说一旦发生狭义上的异常，处理器一定会停止当前操作转而处理异常。但是狭义上的中断则可以被屏蔽掉，RISC-V 架构定义了 CSR 寄存器机器模式中断使能寄存器 mie（Machine Interrupt Enable Registers）可以用于控制中断的屏蔽。

（1）mie 寄存器的详细格式如图 13-8 所示，其中每一个比特域用于控制每个单独的中断使能。

- MEIE 域控制机器模式（Machine Mode）下外部中断（External Interrupt）的屏蔽。

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI	MEIE	WPRI	SEIE	UEIE	MTIE	WPRI	STIE	UTIE	MSIE	WPRI	SSIE	USIE
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

图 13-8 mie 寄存器的格式

- MTIE 域控制机器模式（Machine Mode）下计时器中断（Timer Interrupt）的屏蔽。
- MSIE 域控制机器模式（Machine Mode）下软件中断（Software Interrupt）的屏蔽。

（2）软件可以通过写 mie 寄存器中的值达到屏蔽某些中断的效果。假设 MTIE 域为被设置成 0，则意味着将计时器中断屏蔽，处理器将无法响应计时器中断。

（3）如果处理器（譬如蜂鸟 E200）只实现了机器模式，则监督模式和用户模式对应的中断使能位（SEIE、UEIE、STIE、UTIE、SSIE 和 USIE）无任何意义。

注意：由于本书为简化知识模型，在此仅介绍“只支持机器模式”的架构，因此对 SEIE，UEIE、STIE、UTIE、SSIE 和 USIE 等不做赘述。对其感兴趣的读者请参见 RISC-V“特权架构文档”原文。

注意：除了对 3 种中断的分别屏蔽，通过 mstatus 寄存器中的 MIE 域还可以全局关闭所有中断。

13.3.3 中断等待

RISC-V 架构定义了 CSR 寄存器机器模式中断等待寄存器 mip（Machine Interrupt Pending Registers）可以用于查询中断的等待状态。

（1）mip 寄存器的详细格式如图 13-9 所示，其中的每一个域用于反映每个单独的中断等待状态（Pending）。

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	WIRI	SEIP	UEIP	MTIP	WIRI	STIP	UTIP	MSIP	WIRI	SSIP	USIP
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

图 13-9 mip 寄存器的格式

- MEIP 域反映机器模式（Machine Mode）下的外部中断的等待（Pending）状态。
- MTIP 域反映机器模式（Machine Mode）下的计时器中断的等待（Pending）状态。
- MSIP 域反映机器模式（Machine Mode）下的软件中断的等待（Pending）状态。

（2）如果处理器（譬如蜂鸟 E200）只实现了机器模式，则 mip 寄存器中监督模式和用户模式对应的中断等待状态位（SEIP、UEIP、STIP、UTIP、SSIP 和 USIP）无任何意义。

注意：由于本书为简化知识模型，在此仅介绍“只支持机器模式”的架构，因此对 SEIP、UEIP、STIP、UTIP、SSIP 和 USIP 等不做赘述。对其感兴趣的读者请参见 RISC-V“特权架构文档”原文。

（3）软件可以通过读 mip 寄存器中的值达到查询中断状态的效果。

- 如果 MTIP 域的值为 1，则表示当前有计时器中断（Timer Interrupt）正在等待“Pending”。注意：即便 mie 寄存器中 MTIE 域的值为 0（被屏蔽），如果计时器中断到来，则 MTIP 域仍然能够显示为 1。
- MSIP 和 MEIP 与 MTIP 同理。

（4）MEIP/MTIP/MSIP 域的属性均为只读，软件无法直接写这些域改变其值。只有这些中断的源头被清除后将中断源撤销，MEIP/MTIP/MSIP 域的值才能相应地归零。譬如 MEIP 对应的外部中断需要程序进入中断服务程序后配置外部中断源，将其中断撤销。MTIP 和 MSIP 同理。下一节将详细介绍中断的类型和清除方法。

13.3.4 中断优先级与仲裁

对于中断而言，第 13.1.1 节中曾经提到多个中断可能存在着优先级仲裁的情况。对于 RISC-V 架构而言，分为如下 3 种情况。

（1）如果 3 种中断同时发生，其响应的优先级顺序如下，mcause 寄存器中将按此优先级顺序选择更新异常编号（Exception Code）的值。

- 外部中断（External Interrupt）优先级最高。
- 软件中断（Software Interrupt）其次。
- 计时器中断（Timer Interrupt）再次。

（2）调试中断比较特殊。只有调试器（Debugger）介入调试时才发生，正常情形下不会发生，因此在此不予讨论。关于调试方案的详细信息请参见第 14 章。

（3）由于外部中断来自 PLIC，而 PLIC 可以管理数量众多的外部中断源，多个外部中断源之间的优先级和仲裁可通过配置 PLIC 的寄存器进行管理。请参见附录 C 了解 PLIC 的更多信息。

13.3.5 中断嵌套

第 13.1.1 节中曾经提到多个中断理论上可能存在着中断嵌套的情况。而对于 RISC-V 架构而言，如第 13.2.1 节中所述。

- 进入异常之后，mstatus 寄存器中的 MIE 域将会被硬件自动更新成为 0（意味着中断被全局关闭，从而无法响应新的中断）。
- 退出中断后，MIE 域才被硬件自动恢复成中断发生之前的值（通过 MPIE 域得到），从而再次全局打开中断。

由上可见，一旦响应中断进入异常模式后，中断被全局关闭再也无法响应新的中断，因此 RISC-V 架构定义的硬件机制默认无法支持硬件中断嵌套行为。

如果一定要支持中断嵌套，需要使用软件的方式达到中断嵌套的目的，从理论上来讲，可采用如下方法。

（1）在进入异常之后，软件通过查询 mcause 寄存器确认这是响应中断造成的异常，并跳入相应的中断服务程序中。在这期间，由于 mstatus 寄存器中的 MIE 域被硬件自动更新成为 0，因此新的中断都不会被响应。

（2）待程序跳入中断服务程序中后，软件可以强行改写 mstatus 寄存器的值，而将 MIE 域的值改为 1，意味着将中断再次全局打开。从此时起，处理器将能够再次响应中断。但是在强行打开 MIE 域之前，需要注意如下事项。

- 假设软件希望屏蔽比其优先级低的中断，而仅允许优先级比它高的新来打断当前中断，那么软件需要通过配置 mie 寄存器中的 MEIE/MTIE/MSIE 域，来有选择地屏蔽不同类型的中断。
- 对于 PLIC 管理的众多外部中断而言，由于其优先级受 PLIC 控制，假设软件希望屏蔽比其优先级低的中断，而仅允许优先级比它高的新来中断打断当前中断，那么软件需要通过配置 PLIC 阈值（Threshold）寄存器的方式来有选择地屏蔽不同类型的中断。

（3）在中断嵌套的过程中，软件需要注意保存上下文至存储器堆栈中，或者从存储器堆栈中将上下文恢复（与函数嵌套同理），

（4）在中断嵌套的过程中，软件还需要注意将 mepc 寄存器，以及为了实现软件中断嵌套被修改的其他 CSR 寄存器的值保存至存储器堆栈中，或者从存储器堆栈中恢复（与函数嵌套同理）。

除此之外，RISC-V 架构也允许用户实现自定义的中断控制器实现硬件中断嵌套功能。

13.3.6 总结比较

中断和异常虽说本身不是一种指令，但却是处理器指令集架构非常重要的一环，任何一种指令集架构都会安排专门的章节去定义中断和异常的行为。同时中断和异常也往往是最复杂和难以理解的部分，可以说要了解一门处理器架构，熟悉其中断和异常的处理机制是必不可少的。

对 ARM 的 Cortex-M 系列或者 Cortex-A 系列比较熟悉的读者，可能会了解 Cortex-M 系列定义的嵌套向量中断控制器（Nested Vector Interrupt Controller，NVIC）和 Cortex-A 系列定义的通用中断控制器（General Interrupt Controller，GIC）。这两种中断控制器都非常强大，但也非常复杂。相比而言，RISC-V 架构中断和异常机制则要简单得多，这同样反映了 RISC-V 架构力图简化硬件的设计哲学。

13.4 RISC-V 架构异常相关 CSR 寄存器

将 RISC-V 架构中所有中断和异常相关的寄存器加以总结，如表 13-2 所示。

表 13-2 中断和异常相关的寄存器

类 型	名 称	全 称	描 述
CSR	mtvec	机器模式异常入口基地址寄存器（Machine Trap-Vector Base-Address Register）	定义进入异常的程序 PC 地址
	mcause	机器模式异常原因寄存器（Machine Cause Register）	反映进入异常的原因
	mtval (mbadaddr)	机器模式异常值寄存器（Machine Trap Value Register）	反映进入异常的信息
	mepc	机器模式异常 PC 寄存器（Machine Exception Program Counter）	用于保存异常的返回地址
	mstatus	机器模式状态寄存器（Machine Status Register）	mstatus 寄存器中的 MIE 域和 MPIE 域用于反映中断全局使能
	mie	机器模式中断使能寄存器（Machine Interrupt Enable Registers）	用于控制不同类型中断的局部使能
	mip	机器模式中断等待寄存器（Machine Interrupt Pending Registers）	反映不同类型中断的等待状态
Memory Address Mapped	mtime	机器模式计时器寄存器（Machine-mode timer register）	反映计时器的值
	mtimecmp	机器模式计时器比较值寄存器（Machine-mode timer compare register）	配置计时器的比较值
	msip	机器模式软件中断等待寄存器（Machine-mode Software Interrupt Pending Register）	用以产生或者清除软件中断
	PLIC	PLIC 的所有功能寄存器，请参见附录 C5	

13.5 蜂鸟 E200 异常处理的硬件实现

本节将介绍蜂鸟 E200 处理器对异常处理的硬件实现和源码分析。

13.5.1 蜂鸟 E200 处理器的异常和中断实现要点

蜂鸟 E200 处理器对于异常和中断的硬件实现，要点概述如下。

- 蜂鸟 E200 为“只支持机器模式”架构，且没有实现 MPU 与 MMU（不会产生虚拟地址 Page Fault 相关的异常），因此只支持上一节中描述的 RISC-V 架构中和机器模式相关的异常类型。
- 蜂鸟 E200 只实现了 RISC-V 架构定义的 3 种基本中断类型（软件中断、计时器中断、外部中断），并未实现更多的自定义中断类型。
- 蜂鸟 E200 的 mtvec 寄存器最低位的 MODE 域仅支持模式 0，即所有的异常响应时处理器均跳转到 BASE 域指示的 PC 地址。

13.5.2 蜂鸟 E200 处理器的异常类型

综合上一节所述，蜂鸟 E200 处理器支持的中断和异常类型总结如表 13-3 所示。

表 13-3　蜂鸟 E200 处理器的异常和中断类型

	异常编号（Exception Code）	异常和中断类型	同步/异步	描　述
中断（Interrupts）	3	机器模式软件中断（Machine software interrupt）	精确异步	机器模式软件中断
	7	机器模式计时器中断（Machine timer interrupt）	精确异步	机器模式计时器中断
	11	机器模式外部中断（Machine external interrupt）	精确异步	机器模式外部中断
异常（Exceptions）	0	指令地址非对齐（Instruction address misaligned）	同步	指令 PC 地址非对齐
	1	指令访问错误（Instruction access fault）	同步	取指令访存错误
	2	非法指令（Illegal instruction）	同步	非法指令
	3	断点（Breakpoint）	同步	RISC-V 架构定义了 EBREAK 指令，当处理器执行到该指令时，会发生异常进入异常服务程序。该指令往往用于调试器（Debugger）使用，譬如设置断点
	4	读存储器地址非对齐（Load address misaligned）	同步	Load 指令访存地址非对齐
	5	读存储器访问错误（Load access fault）	非精确异步	Load 指令访存错误
	6	写存储器和 AMO 地址非对齐（Store/AMO address misaligned）	同步	Store 或者 AMO 指令访存地址非对齐

续表

	异常编号（Exception Code）	异常和中断类型	同步/异步	描　述
异常（Exceptions）	7	写存储器和 AMO 访问错误（Store/AMO access fault）	非精确异步	Store 或者 AMO 指令访存错误
	11	机器模式环境调用（Environment call from M-mode）	同步	机器模式下执行 ECALL 指令。RISC-V 架构定义了 ECALL 指令，当处理器执行到该指令时，会发生异常进入异常服务程序。该指令往往供软件使用，强行进入异常模式
	16	EAI 指令写回错误（EAI Instruction Write-Back Error）	非精确异步	RISC-V 架构只定义了异常编号从 0 到 15 的 16 种异常。因此该异常不是 RISC-V 架构定义的标准异常。此异常是用于蜂鸟 E200 的协处理器扩展指令写回错误造成的异常。有关 EAI 协处理器信息请参见第 16 章

13.5.3 蜂鸟 E200 处理器对于 mepc 的处理

在第 13.2.1 节中曾经提及 RISC-V 架构在中断和异常时的返回地址定义（更新 mepc 的值）有细微的差别。在出现中断时，中断返回地址 mepc 被指向下一条尚未执行的指令。在出现异常时，mepc 则指向当前指令，因为当前指令触发了异常。

按照此原则，蜂鸟 E200 处理器核对于 mepc 值的更新原则如下。

- 对于同步异常，mepc 值更新为当前发生异常的指令 PC 值。
- 对于精确异步异常（即中断），mepc 值更新为下一条尚未执行的指令 PC 值。
- 对于非精确异步异常，mepc 值更新为当前发生异常的指令 PC 值。
- 蜂鸟 E200 处理器核实现中同步异常，精确异步异常以及非精确异步异常的分类如表 13-5 所示。

13.5.4 蜂鸟 E200 处理器的中断接口

如图 13-10 所示，在处理器顶层接口中有 4 根中断输入信号，分别是软件中断、计时器中断、外部中断和调试中断。

- SoC 层面的 CLINT 模块产生一根软件中断信号和一根计时器中断信号，通给蜂鸟 E200 处理器核。
- SoC 层面的 PLIC 接入多个外部中断源将其仲裁后生成一根外部中断信号，通给蜂鸟 E200 处理器核。

- SoC 层面的调试模块生成一根调试中断，通给蜂鸟 E200 处理器核。
- 所有的中断信号均由蜂鸟 E200 处理器核的交付模块进行处理。

CLINT、PLIC 以及交付模块的相关硬件实现和源代码分析将在后续章节分别予以介绍。

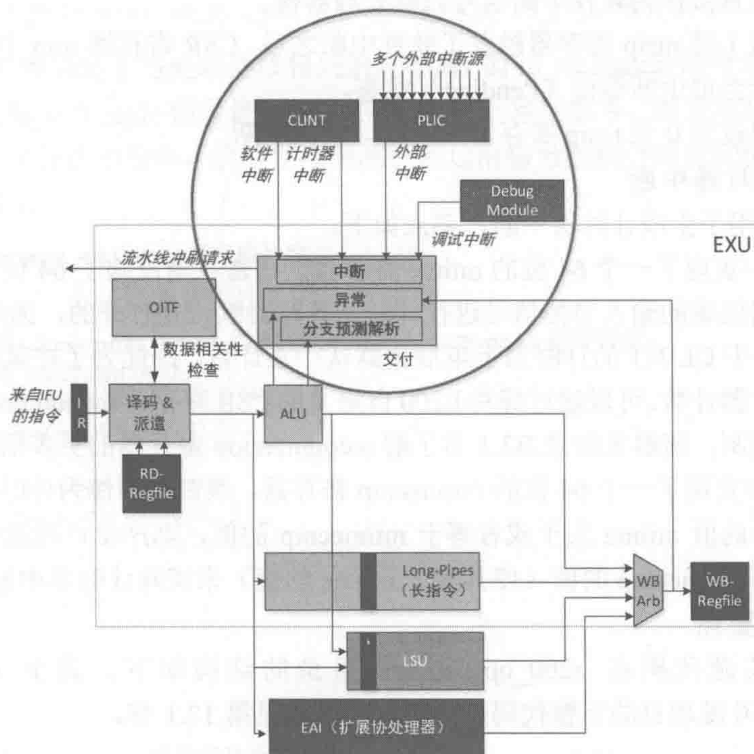


图 13-10　中断和异常处理在流水线中的位置

13.5.5　蜂鸟 E200 处理器 CLINT 微架构及源码分析

CLINT 全称为处理器核局部中断控制器（Core Local Interrupts Controller）。CLINT 是一个存储器地址映射的模块，挂载在处理器核为其实现的专用总线接口上，在蜂鸟 E200 配套的 SoC 中其寄存器的地址区间如表 13-4 所示。注意：CLINT 的寄存器只支持操作尺寸（Size）为 32 位的读写访问。

表 13-4　CLINT 寄存器的存储器映射地址（Memory Mapped Address）

地　址	寄存器名称	功能描述
0x0200_0000	msip	生成软件中断
0x0200_4000	mtimecmp	配置计时器的比较值
0x0200_BFF8	mtime	反映计时器的值

1．生成软件中断

CLINT 可以用于生成软件中断，要点如下。

- CLINT 中实现了一个 32 位的 msip 寄存器。该寄存器只有最低位为有效位，该寄存器有效位直接作为软件中断信号通给处理器核。
- 当软件写 1 至 msip 寄存器触发了软件中断之后，CSR 寄存器 mip 中的 MSIP 域便会置高指示当前中断等待（Pending）状态。
- 软件可通过写 0 至 msip 寄存器来清除该软件中断。

2．生成计时器中断

CLINT 可以用于生成计时器中断，要点如下。

- CLINT 中实现了一个 64 位的 mtime 寄存器。该寄存器反映了 64 位计时器的值。计时器根据低速的输入节拍信号进行计时，计时器默认是打开的，因此会一直计数。
  注意：由于 CLINT 的计时器上电后会默认一直计数，因此为了在某些特殊情况下关闭此计时器计数，可以通过蜂鸟 E200 自定义的 CSR 寄存器 mcounterstop 中的 TIMER 域进行控制，请参见附录 B3.1 节了解 mcounterstop 寄存器的更多信息。
- CLINT 中实现了一个 64 位的 mtimecmp 寄存器。该寄存器作为计时器的比较值，假设计时器的值 mtime 大于或者等于 mtimecmp 的值，则产生计时器中断。软件可以通过改写 mtimecmp 的值（使其大于 mtime 的值）来清除计时器中断。

3．源代码要点

CLINT 相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                  // 存放 RTL 的目录
         |----e203                            // E203 核和 SoC 的 RTL 目录
              |----subsys                     // 存放子系统的 RTL 代码
                   |----e203_subsys_clint.v   // CLINT 的源代码例化模块
              |----perips                     // 存放外设的 RTL 代码
                   |----e203_clint_top.v      // CLINT 的源代码顶层模块
                   |----e203_clint.v          // CLINT 的源代码模块
                   |----sirv_aon_wrapper.v    // Always-on 模块的源代码
```

注意：蜂鸟 E200 的 SoC 系统是基于 SiFive 公司开源的 Freedom E310 SoC 二次开发所得。因此 e203_clint.v 模块来自 Freedom E310 SoC，其代码为高级 Chisel 语言编译生成的 Verilog 代码。有关 SiFive 公司开源 Freedom E310 SoC 与 Chisel 语言的简介，请参见第 3.1.1 节和 3.1.3 节。

CLINT 顶层模块 e203_clint_top 有一根低速的输入节拍信号 io_rtcToggle。该信号来自 SoC 中低速的电源常开域（PowerAlways On Domain）的 io_rtc 信号，因此 io_rtcToggle 的翻转频率与低速时钟的频率一致。相关源代码片段如下所示。

```
// sirv_aon_wrapper.v 源代码片段

  // io_rtc 的翻转频率受低速时钟 aon_clock 控制。
  wire io_rtc_nxt = ~io_rtc;
  wire aon_rst_n = ~aon_reset;
  sirv_gnrl_dffr #(1) io_rtc_dffr (io_rtc_nxt, io_rtc, aon_clock, aon_rst_n);
```

由于 CLINT 模块处于与处理器核相同的时钟域，而 io_rtcToggle 信号来自低速的电源常开时钟域，因此 io_rtcToggle 信号进入 CLINT 模块属于异步信号，需要对其进行同步，然后对同步后的信号进行边沿检测，然后使用探测到的边沿信号控制计时器的值增加一。相关源代码片段如下所示。

```
// e203_subsys_clint.v 源代码片段


    // 使用 sirv_gnrl_sync 同步模块对 aon_rtcToggle 进行同步。
  wire aon_rtcToggle_r;
  sirv_gnrl_sync # (
.DP('E203_ASYNC_FF_LEVELS),
.DW(1)
  ) u_aon_rtctoggle_sync(
.din_a     (aon_rtcToggle_a),
.dout      (aon_rtcToggle_r),
.clk       (clk  ),
.rst_n     (rst_n)
  );


// sirv_clint_top.v 源代码片段

   //将 io_rtcToggle 信号寄存一拍。
  wire io_rtcToggle_r;
   sirv_gnrl_dffr #(1) io_rtcToggle_dffr (io_rtcToggle, io_rtcToggle_r, clk,
rst_n);
    //通过将 io_rtcToggle 信号与寄存后的 io_rtcToggle_r 进行异或计算，从而探测
      // 出 io_rtcToggle 信号的边沿。
  wire io_rtcToggle_edge = io_rtcToggle ^ io_rtcToggle_r;
  wire io_rtcTick = io_rtcToggle_edge;


// sirv_clint.v 源代码片段

// 该代码为 Chisel 编译生成所得，因此属于机器生成代码，人工可读性比较差。

   //io_rtcTick 信号指示 io_rtcToggle 信号的边沿。
```

```
  assign T_904 = {time_1,time_0};
  assign T_906 = T_904 + 64'h1;//计时器按照 io_rtcTick 信号的脉冲进行自增加 1
  assign T_907 = T_906[63:0];
  assign T_909 = T_907[63:32];
......
  assign GEN_6 = io_rtcTick ?T_907 : {{32'd0}, time_0};
......
  assign GEN_10 = T_1280 ? {{32'd0}, T_1015_bits_data} :GEN_6;
......
  always @(posedge clock or posedge reset) begin
    if (reset) begin
      time_0 <= 32'h0;
    end else begin
      time_0 <= GEN_10[31:0];//计时器的低 32 位寄存器
    end
  end

  always @(posedge clock or posedge reset) begin
    if (reset) begin
      time_1 <= 32'h0;
    end else begin
      if (T_1320) begin
        time_1 <= T_1015_bits_data;//计时器的值也可以被软件改写
      end else begin
        if (io_rtcTick) begin
          time_1 <= T_909; //计时器的高 32 位寄存器
        end
      end
    end
  end
```

以上仅对最关键的代码片段予以讲解分析，完整源代码请读者于 GitHub 上的 e200_opensource 项目中自行阅读。

13.5.6 蜂鸟 E200 处理器 PLIC 微架构及源码分析

PLIC 全称为平台级别中断控制器（Platform Level Interrupt Controller），它是 RISC-V 架构标准定义的系统中断控制器，主要用于多个外部中断源的优先级仲裁和派发。关于 PLIC 的详情，请参见附录 C。

PLIC 是一个存储器地址映射的模块，挂载在处理器核为其实现的专用总线接口上，在蜂鸟 E200 配套的 SoC 中其寄存器的地址区间如表 13-5 所示。PLIC 的寄存器只支持 size 为 32 位的读写访问。

表 13-5　PLIC 寄存器的存储器映射地址

地　址	寄存器英文名称	寄存器中文名称
0x0C00_0004	Source 1 priority	中断源 1 的优先级
0x0C00_0008	Source 2 priority	中断源 2 的优先级

续表

地　址	寄存器英文名称	寄存器中文名称
……	……	……
0x0C00_0FFC	Source 1023 priority	中断源 1023 的优先级
……	……	……
0x0C00_1000	Start of pending array（read-only）	中断等待标志的起始地址
……	……	……
0x0C00_107C	End of pending array	中断等待标志的结束地址
……	……	……
0x0C00_2000	Target 0 enables	中断目标 0 的使能位
……	……	……
0x0C20_0000	Target 0 priority threshold	中断目标 0 的优先级门槛
0x0C20_0004	Target 0 claim/complete	中断目标 0 的响应/完成

注意：

- 该 PLIC 理论上可以支持 1024 个中断源，所以此表定义了 1024 个优先级（priority）寄存器的地址和 1024 位等待阵列（pending array）寄存器的地址。但是目前在蜂鸟 E200 SoC 的 PLIC 中，实际只使用到了表 13-6 中的中断源
- 该 PLIC 理论上可以支持多个中断目标（Target）。由于蜂鸟 E200 处理器是一个单核处理器，且仅实现了机器模式，因此仅用到 PLIC 的 Target0，表中的 Target0 即为蜂鸟 E200 处理器核
- 由于蜂鸟 E200 的 PLIC 在 SoC 中的地址映射与 SiFive 公司开源的 Freedom E310 SoC 中 PLIC 寄存器地址映射表完全一致。请参见附录 C5 中关于 Freedom E310 SoC 中 PLIC 的寄存器地址映射与各配置寄存器功能的详细介绍

PLIC 理论上可以支持高达 1024 个外部中断源，在具体的 SoC 中连接的中断源个数可以不一样。蜂鸟 E200 的 SoC 系统基于 SiFive 公司开源的 Freedom E310 SoC 开发所得。PLIC 在此 SoC 中连接了 GIPO、UART、PWM 等多个外部中断源，其中断分配如表 13-6 所示。PLIC 将多个外部中断源仲裁为一个单比特的中断信号送入蜂鸟 E200 处理器核。请参见第 18.2 节了解更多 SoC 的详细信息。

表 13-6　PLIC 的中断分配

PLIC 源中断号	来　源
0	预留为表示没有中断
1	wdogcmp
2	rtccmp
3	uart0
4	uart1
5	qspi0
6	qspi1
7	qspi2
8	gpio0
……	……
39	gpio31

续表

PLIC 源中断号	来　源
40	pwm0cmp0
……	……
43	pwm0cmp3
44	pwm1cmp0
……	……
47	pwm1cmp3
48	pwm2cmp0
……	……
51	pwm2cmp3

PLIC 的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                       // 存放RTL的目录
         |----e203                                 // E203核和SoC的RTL目录
              |----subsys                          // 存放子系统的RTL代码
                   |----e203_subsys_plic.v //  PLIC的源代码例化模块
              |----perips                          // 存放外设的RTL代码
                   |----e203_plic_top.v     // PLIC的源代码顶层模块
                   |----e203_plic_main.v    // PLIC的源代码模块
```

由于 PLIC 模块处于与处理器核相同的时钟域，PLIC 连接的大多数中断来源的设备与 PLIC 处于同一个时钟域，而 RTC 和 WatchDog 中断则来自低速的电源常开时钟域，需要特别对其进行同步。请参见第 18.2 节了解更多 SoC 的详细信息。相关源代码片段如下所示。

```
  // e203_subsys_plic.v源代码片段

output plic_ext_irq,// PLIC最后仲裁所得的一根输出信号作为外部中断通给处理器核。

......

  // 使用sirv_gnrl_sync同步模块对rtc_irq_a和wdg_irq_a进行同步。
wire  wdg_irq_r;
wire  rtc_irq_r;

  sirv_gnrl_sync # (
.DP('E203_ASYNC_FF_LEVELS),
.DW(1)
  ) u_rtc_irq_sync(
.din_a     (rtc_irq_a),
.dout      (rtc_irq_r),
.clk       (clk  ),
.rst_n     (rst_n)
```

```
    );

    sirv_gnrl_sync # (
  .DP('E203_ASYNC_FF_LEVELS),
  .DW(1)
    ) u_wdg_irq_sync(
  .din_a     (wdg_irq_a),
  .dout      (wdg_irq_r),
  .clk       (clk  ),
  .rst_n     (rst_n)
    );

  ……

      // 分配多个外部中断源作为 PLIC 的输入。
    wire plic_irq_i_0  = wdg_irq_r; //来自 WatchDog 模块的中断
    wire plic_irq_i_1  = rtc_irq_r; //来自 RTC 模块的中断
    wire plic_irq_i_2  = uart0_irq;
    wire plic_irq_i_3  = uart1_irq;
    wire plic_irq_i_4  = qspi0_irq;
    wire plic_irq_i_5  = qspi1_irq;
    wire plic_irq_i_6  = qspi2_irq;
    wire plic_irq_i_7  = gpio_irq_0 ;
    wire plic_irq_i_8  = gpio_irq_1 ;
    wire plic_irq_i_9  = gpio_irq_2 ;
    wire plic_irq_i_10 = gpio_irq_3 ;
    wire plic_irq_i_11 = gpio_irq_4 ;
    wire plic_irq_i_12 = gpio_irq_5 ;
    wire plic_irq_i_13 = gpio_irq_6 ;
    wire plic_irq_i_14 = gpio_irq_7 ;
    wire plic_irq_i_15 = gpio_irq_8 ;
    wire plic_irq_i_16 = gpio_irq_9 ;
    wire plic_irq_i_17 = gpio_irq_10;
    wire plic_irq_i_18 = gpio_irq_11;
    wire plic_irq_i_19 = gpio_irq_12;
    wire plic_irq_i_20 = gpio_irq_13;
    wire plic_irq_i_21 = gpio_irq_14;
    wire plic_irq_i_22 = gpio_irq_15;
    wire plic_irq_i_23 = gpio_irq_16;
    wire plic_irq_i_24 = gpio_irq_17;
    wire plic_irq_i_25 = gpio_irq_18;
    wire plic_irq_i_26 = gpio_irq_19;
    wire plic_irq_i_27 = gpio_irq_20;
    wire plic_irq_i_28 = gpio_irq_21;
    wire plic_irq_i_29 = gpio_irq_22;
    wire plic_irq_i_30 = gpio_irq_23;
    wire plic_irq_i_31 = gpio_irq_24;
    wire plic_irq_i_32 = gpio_irq_25;
    wire plic_irq_i_33 = gpio_irq_26;
    wire plic_irq_i_34 = gpio_irq_27;
    wire plic_irq_i_35 = gpio_irq_28;
    wire plic_irq_i_36 = gpio_irq_29;
    wire plic_irq_i_37 = gpio_irq_30;
```

```
wire plic_irq_i_38 = gpio_irq_31;
wire plic_irq_i_39 = pwm0_irq_0;
wire plic_irq_i_40 = pwm0_irq_1;
wire plic_irq_i_41 = pwm0_irq_2;
wire plic_irq_i_42 = pwm0_irq_3;
wire plic_irq_i_43 = pwm1_irq_0;
wire plic_irq_i_44 = pwm1_irq_1;
wire plic_irq_i_45 = pwm1_irq_2;
wire plic_irq_i_46 = pwm1_irq_3;
wire plic_irq_i_47 = pwm2_irq_0;
wire plic_irq_i_48 = pwm2_irq_1;
wire plic_irq_i_49 = pwm2_irq_2;
wire plic_irq_i_50 = pwm2_irq_3;
```

以上仅对关键的代码片段予以讲解分析，完整源代码请读者到 GitHub 上的 e200_opensource 项目中自行阅读。

13.5.7 蜂鸟 E200 处理器交付模块对中断和异常的处理

在第 9 章中介绍过交付模块是指令的交付点，一条指令一旦被交付，则意味着它真正得到了执行。因此蜂鸟 E200 的中断和异常处理均在交付模块中进行处理，如图 13-10 中圆圈处所示。

1. 异常的处理

交付模块对于异常处理部分的要点如下。

- 交付模块接受来自 ALU 的交付请求，对于每一条 ALU 执行的指令，可能发生异常。如果没有发生异常，则该指令顺利交付；如果发生了异常，则会造成流水线冲刷（Pipeline Flush）。ALU 指令造成的异常均为同步异常，如表 13-3 所示，同步异常均来自于 ALU 模块。同步异常能够准确地定位于当前正在执行的 ALU 指令，因此 mepc 寄存器中更新的 PC 值即为当前正在交付指令（来自于 ALU 接口）的 PC。
- 交付模块接受来自长指令写回时的交付请求，每一条长指令可能发生异常。如果没有发生异常，则该指令顺利交付；如果发生了异常，则会造成流水线冲刷（Pipeline Flush）。长指令写回的异常均为非精确异步异常，如表 13-3 所示，非精确异步异常均来自长指令写回时的请求。

长指令异常的“异常返回地址”将会使用此指令自己的 PC，即 mepc 寄存器中更新的 PC 值为此指令的 PC。但是由于这长指令可能已经被交付了，若干个周期过去了，且在这若干周期内可能又有新的后续指令已经写回了通用寄存器组，因此其响应异常后的处理器状态是一种非精确状态（无法定义为某一条指令的边界），属于非精确异步异常。

2. 中断的处理

交付模块对于中断处理部分的要点如下。

- 交付模块接受来自 CLINT 和 PLIC 的 3 根中断信号的请求，蜂鸟 E200 的实现中将中断作为一种精确异步异常，这种异常的“异常返回地址”将会为下一条尚未交付的指令，即 mepc 寄存器中更新的 PC 值即为下一条待交付的指令（来自于 ALU 接口）的 PC。
- 当异步异常和 ALU 造成的同步异常以及中断同时发生时，优先级依次为：长指令造成的异步异常优先级最高，中断造成的异步异常其次，ALU 造成的同步异常最后。
- 异常一旦发生，便会冲刷流水线，将后续的指令取消掉，并向 IFU 模块发送冲刷请求（Flush Request）和重新取指令的 PC（称之为 Flush PC），用以重新从新的 PC 地址开始取指令。
- 特殊的调试中断也在交付模块中处理，本章不做介绍，请参见第 14 章了解调试相关的信息。

3. 源代码要点

蜂鸟 E200 处理器核中断和异常处理的相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                 // 存放RTL的目录
         |----e203                           // E203核和SoC的RTL目录
              |----general                   // 存放一些公用的通用RTL代码
              |----core                      // 存放e203 Core的RTL代码
                   |----e203_exu_commit.v        //交付模块顶层
                   |----e203_exu_excp.v          //交付模块中处理
                                             //  异常和中断的子模块
```

交付模块中的中断和异常处理相关源代码片段如下所示。

```
    // e203_exu_excp.v 源代码片段

    ……

    // 生成冲刷请求，包括长指令造成的异常、调试中断造成的异常、普通中断造成的异常和ALU指令造成的异常。

    assign excpirq_flush_req  = longp_excp_flush_req | dbg_entry_flush_req | irq_flush_req | alu_excp_flush_req;

    ……

    // 生成重新取指令的PC（Flush PC），只要不是调试中断造成的冲刷，都会使用CSR寄存器mtvec中的值。
    assign excpirq_flush_pc = dbg_entry_flush_req ? 'E200_PC_SIZE'h800 : (all_excp_flush_req & dbg_mode) ? 'E200_PC_SIZE'h808 : csr_mtvec_r;

    ……
```

```
 // 根据中断的类型，更新 mcause 寄存器中的异常编号（Exception Code）
   assign irq_cause[31] = 1'b1;
   assign irq_cause[30:4] = 27'b0;
   assign irq_cause[3:0]  =  sft_irq_r ? 4'd3  :// 3  Machine software interrupt
                             tmr_irq_r ? 4'd7  :// 7  Machine timer interrupt
                             ext_irq_r ? 4'd11 :// 11 Machine external interrupt
                                         4'b0;
……

 // 根据异常的类型，更新 mcause 寄存器中的异常编号

   wire ['E200_XLEN-1:0] excp_cause;
   assign excp_cause[31:5] = 27'b0;
   assign excp_cause[4:0]  =
       alu_excp_flush_req_ifu_misalgn? 5'd0 //Instruction address misaligned
     : alu_excp_flush_req_ifu_buserr ? 5'd1 //Instruction access fault
     : alu_excp_flush_req_ifu_ilegl  ? 5'd2 //Illegal instruction
     : alu_excp_flush_req_ebreak     ? 5'd3 //Breakpoint
     : alu_excp_flush_req_ld_misalgn ? 5'd4 //load address misalign
     : (longp_excp_flush_req_ld_buserr | alu_excp_flush_req_ld_buserr) ? 5'd
5 //load access fault
     : alu_excp_flush_req_stamo_misalgn ? 5'd6 //Store/AMO address misalign
     : (longp_excp_flush_req_st_buserr | alu_excp_flush_req_stamo_buserr) ?
5'd7 //Store/AMO access fault
     : (alu_excp_flush_req_ecall & u_mode) ? 5'd8 //Environment call from U-
mode
     : (alu_excp_flush_req_ecall & s_mode) ? 5'd9 //Environment call from S-
mode
     : (alu_excp_flush_req_ecall & h_mode) ? 5'd10 //Environment call from H
-mode
     : (alu_excp_flush_req_ecall & m_mode) ? 5'd11 //Environment call from M
-mode
     : longp_excp_flush_req_insterr ? 5'd16// This only happened for the EAI
 long instructions actually
   : 5'h1F;//Otherwise a reserved value
  ……

  assign cmt_cause = excp_taken_ena ? excp_cause : irq_cause;

  ……
  // 对于长指令，使用其自身的指令 PC 值。
  // 对于普通 ALU 指令，使用当前交付接口（来自于 ALU 接口）的指令 PC 更新 mepc 寄存器。
  assign cmt_epc = longp_excp_i_valid ? longp_excp_i_pc : alu_excp_i_pc;
```

e203_exu_excp 模块中的内容非常琐碎繁杂，必须了解 RISC-V 架构的很多细节才能理解。以上仅对关键的代码片段予以讲解分析，完整源代码请读者到 GitHub 上的 e200_opensource 项目中自行阅读。

4. mret 指令的处理

mret 指令会触发处理器退出异常模式。该指令在蜂鸟 E200 处理器中被当作一种跳转指令来执行，其硬件实现与第 9 章中描述的分支指令解析一样在 e203_exu_branchslv 模块中处理。

```
e200_opensource
    |----rtl                              // 存放 RTL 的目录
        |----e203                         // E203 核和 SoC 的 RTL 目录
            |----general                  // 存放一些公用的通用 RTL 代码
            |----core                     // 存放 e203 Core 的 RTL 代码
                |----e203_exu_commit.v         //交付模块顶层
                |----e203_exu_branchslv.v      //交付中处理
                                          //  mret 指令的子模块
```

相关源代码片段如下所示。

```
    // e200_exu_branchslv.v 源代码片段

 // 生成冲刷请求，包括了 mret 指令。

  wire brchmis_need_flush = (
       (cmt_i_bjp & (cmt_i_bjp_prdt ^ cmt_i_bjp_rslv))
  //   If it is a FenceI instruction, it is always jump
       | cmt_i_fencei
  //   If it is a RET instruction, it is always jump
     | cmt_i_mret
  //   If it is a DRET instruction, it is always jump
 | cmt_i_dret
     );

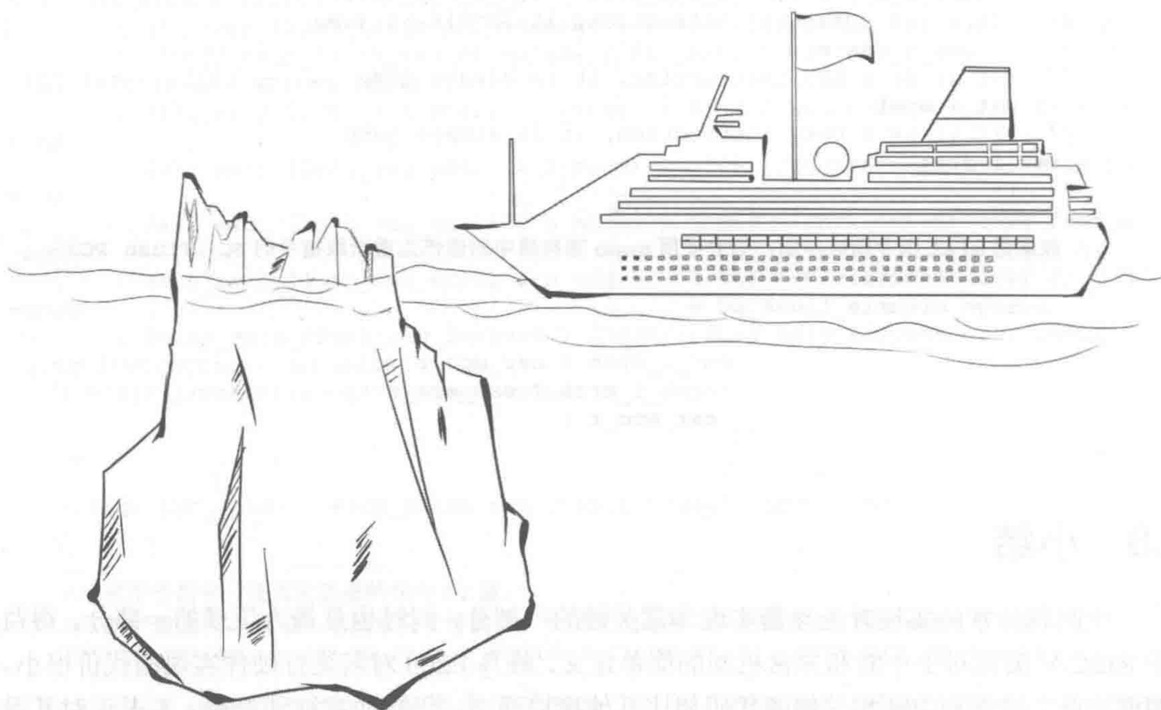
// 如果是 mret 指令造成冲刷，则会使用 mepc 寄存器中的值作为重新取指令的 PC（Flush PC）。

   assign brchmis_flush_pc =
……
                          cmt_i_dret ? csr_dpc_r :
                          //cmt_i_mret ? csr_epc_r :
                           csr_epc_r ;
……
```

13.5.8 小结

中断和异常的实现时处理器实现非常关键的一部分，同时也是最为烦琐的一部分。得益于 RISC-V 架构对于中断和异常机制的简单定义，蜂鸟 E200 对其进行硬件实现的代价很小。即便如此，异常和中断相关的源代码相比其他模块而言，仍然非常细琐繁杂，本书仅对其设计要点以及代码片段进行简要地讲解分析，请感兴趣的读者到 GitHub 上的 e200_opensource 项目中自行阅读完整源代码。

第 14 章　最不起眼的，其实是最难的——调试机制



对于一款处理器而言，人们在研究其微架构时往往关注的是聚光灯下的某些特性，譬如流水线的级数、运算单元的能力等，而对于角落里的另外一位“小朋友”往往未加重视，这位“小朋友”便是调试（Debug）单元。

不同于普通的ASIC芯片，处理器运行的是软件程序。试想一下，如果一款处理器不具备调试能力，那么一旦程序运行出现问题，开发人员便束手无策，处理器也就秒变为“砖”了。因此，处理器对于运行于其上的软件程序提供调试能力是至关重要的。

调试单元在处理器设计中往往是幕后英雄，大多数人对其软硬件实现机制也是不明就里，或者根本未曾关注。但是，最不起眼的，往往是最难的，调试机制是个非常复杂的软硬件协作机制，软硬件的实现难度很大。

目前绝大多数开源处理器仅提供处理器核的实现，并没有提供调试方案的实现，很少有开源处理器能够支持完整的GDB交互调试功能。蜂鸟E200不仅开源了处理器核的实现、SoC实现、FPGA平台和软件示例，还实现和开源了完整的调试方案，具备完整的GDB交互调试功能。可以说是从硬件到软件，从模块到SoC，从运行到调试的一套完整解决方案。

本章将带着读者一探RISC-V调试机制的究竟，同时结合蜂鸟E200处理器实例来简述调试机制的硬件实现。值得再次强调的是，调试机制和软硬件实现是一个非常完整且复杂的议题，若要将其彻底阐述清楚，几乎可以单独成书，本书本章只能以极其有限的篇幅揭开其冰山一角。有兴趣的读者可以根据本章推荐的文档自行仔细研究，也可仔细研究蜂鸟E200调试单元相关的Verilog源代码。

14.1 调试机制概述

对于处理器的调试功能而言，常用的两种是“交互式调试”和“追踪调试”。本节将对此两种调试的功能及原理加以简述。

14.1.1 交互调试概述

交互调试（Interactive Debug）功能是处理器提供的最常见的一种调试功能，从最低端的处理器到最高端的处理器，交互调试几乎是必备的功能。交互调试是指调试器软件（譬如常见的调试软件GDB）能够直接对处理器取得控制权，进而对其以一种交互的方式进行调试，譬如通过调试软件对处理器。

- 下载或者启动程序。
- 通过设定各种特定条件来停止程序。
- 查看处理器的运行状态。包括通用寄存器的值、存储器地址的值等。

- 查看程序的状态。包括变量的值、函数的状态等。
- 改变处理器的运行状态。包括通用寄存器的值、存储器地址的值等。
- 改变程序的状态。包括变量的值、函数的状态等。

有关 GDB 的简介和如何运行 GDB 软件进行调试，请参见第 19.4 节。

对于嵌入式平台而言，调试器软件一般是运行于主机 PC 端的一款软件，而被调试的处理器往往是在嵌入式开发板之上，这是交叉编译和远程调试的一种典型情形。调试器软件为何能够取得处理器的控制权，从而对其进行调试呢？可想而知，需要硬件的支持才能做到。在处理器核的硬件中，往往需要一个硬件调试模块。该调试模块通过物理介质（譬如 JTAG 接口）与主机端的调试软件进行通信接受其控制，然后调试模块对处理器核进行控制。

为了帮助读者进一步理解，以交互式调试中常见的一种调试情形为例来阐述此过程。假设调试软件 GDB 试图对程序中的某个 PC 地址设置一个断点，然后希望程序运行到此处之后停下来，之后 GDB 能够读取处理器当时的某个寄存器的值。调试软件和调试模块便会进行如下协同操作。

- 开发人员通过运行于主机端的 GDB 软件在其软件界面上设置某行程序的断点，GDB 软件通过底层驱动 JTAG 接口访问远程处理器的调试模块，对其下达命令，告诉其希望于某 PC 设置一个断点。
- 调试模块得令即开始对处理器核进行控制，首先它会请求处理器核停止；然后修改存储器中那个 PC 地址的指令，将其替换成一个 Breakpoint 指令；最后将处理器核放行，让处理器恢复执行。
- 当处理器恢复执行后，执行到那个 PC 地址时，由于碰到了 Breakpoint 指令，会产生异常进入调试模式的异常服务程序。调试模块探测到处理器核进入了调试模式的异常服务程序，并将此信息显示出来。主机端的 GDB 软件一直在监测调试模块的状态从而得知此信息，便得知处理器核已经运行到断点处停止了下来，并显示在 GDB 软件界面上。
- 开发人员通过运行于主机端的 GDB 软件在其软件界面上设置读取某个寄存器的值，GDB 软件通过底层驱动 JTAG 接口访问远程处理器的调试模块，对其下达命令，告诉其希望读取某个寄存器的值。
- 调试模块得令即开始对处理器核进行控制，从处理器核中将那个寄存器的值读取出来，并将此信息显示出来。主机端的 GDB 软件一直在监测调试模块的状态，从而得知此信息，便通过 JTAG 接口将读取的值返回到主机 PC 端，并显示在 GDB 软件界面上。

注意：以上采用极为通俗的语言来描述此过程，以帮助读者理解，但难免失之严谨，请以具体的调试机制文档为准。

从上述过程中可以看出，调试机制是一套复杂的软硬件协同工作机制，需要调试软件和硬件调试模块的精密配合。

同时，也可以看出交互式调试对于处理器的运行往往是具有打扰性（Intrusive）的。调试单元会在后台偷偷地控制住处理器核，时而让其停止，时而让其运行。由于交互式调试对处理器运行的程序具有影响，甚至会改变其行为，尤其是对时间先后性有依赖的程序，有时候交互式调试并不能完整地重现某些程序的 Bug。最常见的情形便是处理器在全速运行某个程序时会出现 Bug，当开发人员使用调试软件对其进行交互式调试时，Bug 又不见了。如此反复，好不折煞人也。其主要原因往往就是交互式调试过程的打扰性（Intrusive），使得程序在调试模式和全速运行下的结果出现了差异。

14.1.2 跟踪调试概述

上一节中论述了交互式调试的一个缺点是对处理器的运行具有打扰性，为了克服此种缺陷，便引入了跟踪调试（Trace Debug）机制。

跟踪调试，即调试器只跟踪记录处理器核执行过的所有程序指令，而不会打断干扰处理器的执行过程。跟踪调试同样需要硬件的支持才能做到，相比交互式调试的实现难度更大。由于处理器的运行速度非常快，每秒钟能执行上百万条指令，如果长时间运行某个程序，其产生的信息量十分巨大。跟踪调试器的硬件单元需要跟踪记录下所有的指令，对于处理速度的要求，数据的压缩、传输和存储等都是极大挑战。跟踪调试器的硬件实现会涉及相比交互调试而言更加复杂的技术，同时硬件开销也更大，因此跟踪调试器往往只在比较高端的处理器中使用。

14.2 RISC-V 架构的调试机制

由于处理器可以有不同的调试（Debug）实现机制，且与微架构的实现有关，譬如有的调试机制追求以较小的面积实现调试的功能，有的调试机制追求以较快的调试反应速度而付出较大的面积开销。因此 RISC-V 基金会目前还没有发布标准的 RISC-V 调试架构文档（RISC-V Debug Specification），但是有若干公开的候选文档（Proposal Specifications）在审核之中。

目前公开的候选调试架构文档（RISC-V Debug Proposal Specifications）中，比较有影响力当属 SiFive 公司的方案，如图 14-1 所示。并且 SiFive 公司基于此方案实现了开源的 Freedom E310 SoC，基于此 SoC 实际流片量产的芯片和 HiFive1 开发板被广泛应用（仅支持交互式调试）。因此 SiFive 公司的调试架构文档具有相当高的可信度。

读者可以从 SiFive 公司网站下载其文档。其网站上有 0.11 和 0.13 两个版本，开源的 Freedom E310 SoC 和 HiFive1 开发板使用的是 0.11 版本（riscv-debug-spec-0.11nov12.pdf），本章将其简称为“0.11 版本”。

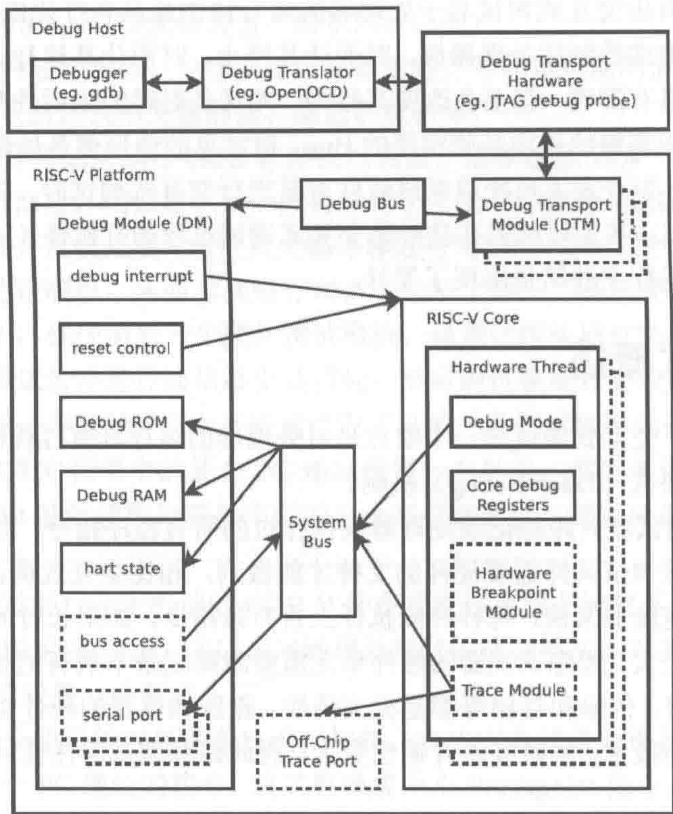


图 14-1　SiFive 公司的 RISC-V 系统调试方案图

14.2.1 调试器软件的实现

如第 14.1 节中所述，完整的调试机制需要调试器软件（Debugger，譬如 GDB）和硬件精密协作。在第 14.1 节中曾列举了软硬件如何精密配合，达到向程序中设置断点和读取寄存器的通俗示例。感兴趣的读者若要试图严谨地理解调试机制，并结合调试器软件和硬件调试模块（Debug Module）通过软硬件精密协作的方式实现所有的调试功能，可以参见“0.11 版本”原文。

14.2.2 调试模式

“0.11 版本”中定义了一种特殊的处理器模式——调试模式（Debug Mode），同时定义了若干种触发条件，处理器核一旦遇到此类触发条件便会进入调试模式。可以将进入调试模式当成一种特殊的异常，进入调试模式时，处理器核会进行如下更新。

- 处理器 PC 跳转到 0x800 地址。
- 将处理器正在执行的指令 PC 保存到 CSR 寄存器 dpc 中。
- 将引发进入调试模式的触发原因保存到 CSR 寄存器 dcsr 中。

关于调试模式的详细信息，请参见“0.11 版本”原文。

14.2.3 调试指令

RISC-V 标准指令集中定义了一条特殊的断点指令 ebreak，此指令主要用于调试软件设置断点。当处理器核执行到这条指令时会跳转到异常模式或者调试模式。

“0.11 版本”中还定义了一条特殊的指令 dret（注意和 mret 区分）。dret 指令执行后，处理器核会进行如下更新。

- 处理器 PC 跳转到保存在 dpc 中的值，这意味着处理器退回到之前进入调试模式之前的程序执行点。
- 将 dcsr 寄存器中的域清除掉，指示处理器退出了调试模式。

14.2.4 调试机制 CSR

“0.11 版本”中定义了几个只能在调试模式下访问到的 CSR 寄存器。请参见“0.11 版本”原文了解其详情。

14.2.5 调试中断

“0.11 版本”中定义了一个特殊的处理器中断类型——调试中断（Debug Interrupt）。当处理器核收到此中断之后，将进入调试模式。调试中断是最主要的进入调试模式的触发条件，挑调试器软件的众多功能均依赖于此中断。

关于调试中断的详细信息，请参见“0.11 版本”原文。

14.3 蜂鸟 E200 调试机制的硬件实现

蜂鸟 E200 处理器核调试机制的硬件实现严格依据“0.11 版本”定义的方案。与开源的 HiFive1 芯片一样，蜂鸟 E200 处理器目前仅支持交互式调试，尚不支持追踪调试。

14.3.1 蜂鸟 E200 交互式调试概述

蜂鸟 E200 处理器核交互式调试机制的硬件实现要点如下。

- 如图 14-1 所示，调试主机（Debug Host）为主机 PC 端的调试平台，由于嵌入式系统往往以交叉编译远程调试的方式工作，因此软件的开发编译在主机 PC 端完成，并且在主机 PC 端运行调试软件。譬如常用的 GDB 调试软件，对嵌入式硬件平台（譬如基于 RISC-V 的 MCU）进行调试。有关如何在主机 PC 端进行交叉编译和如何运行 GDB 软件进行远程调试，请参见第 19.4 节。
- 主机 PC 端的 GDB 软件需要与其 Gdbserver 通信，Gdbserver 可以用开源软件 OpenOCD 充当。OpenOCD 的源代码中包含了各种常见硬件芯片的驱动，譬如 FTDI 公司的 USB 转 JTAG 芯片。因此此芯片的 USB 接口可以使用 USB 连接线与主机 PC 连接，此芯片的 JTAG 接口则可以与 RISC-V 处理器的 SoC 硬件平台相连。其原理图和实物对照如图 14-2 所示。有关 OpenOCD 软件的介绍和使用请参见第 19.4 节。有关实物图中使用到的具体器材型号和手工制作过程请参见第 18.3.3 节。
- 如图 14-2 所示，在 RISC-V 的 SoC 中，JTAG 接口由 DTM 模块转换成为内部的调试总线，通过该总线访问调试模块。DTM 和调试模块在后续章节中另行论述。

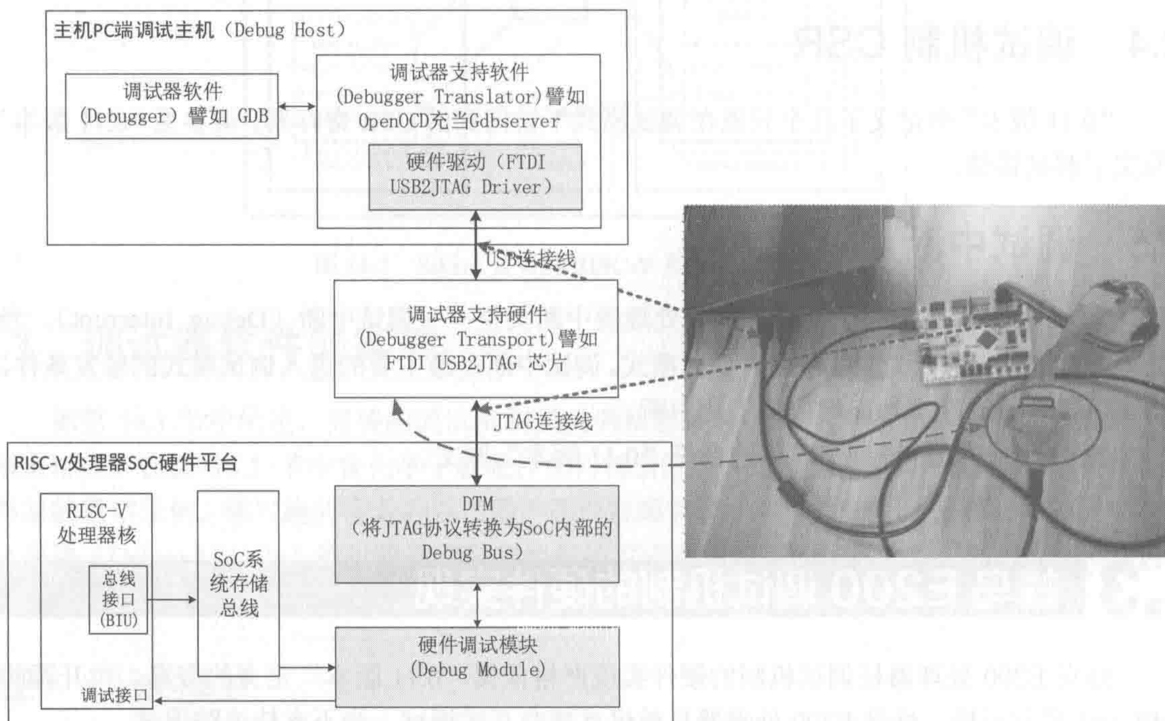


图 14-2 蜂鸟 E200 处理器的 Debug 机制原理与实物对照图

14.3.2 DTM 模块

如图 14-1 所示，DTM 全称 Debug Transport Module，在蜂鸟 E200 处理器中 DTM 主要是将 JTAG 标准接口转换成为内部的调试总线（Debug Bus），其硬件实现要点如下。

- DTM 源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                    // 存放 RTL 的目录
        |----e203                               // E203 核和 SoC 的 RTL 目录
            |----debug                          // 存放调试相关模块的 RTL 代码
                |----sirv_jtag_dtm.v   // DTM 模块
```

- 该模块主要是使用状态机对 JTAG 协议进行解析转换成调试总线。且由于 DTM 模块处于 JTAG 时钟域，与调试总线要访问的调试模块不属于同一个时钟域，因此需要被同步。具体代码请读者到 GitHub 的 e200_opensource 项目中自行阅读。

14.3.3 硬件调试模块

如图 14-1 所示，硬件调试模块在整个调试机制担任了最重要的硬件角色，其硬件实现要点如下。

- 调试模块相关源代码在 e200_opensource 目录的结构如下。关于 GitHub 网站上 e200_opensource 开源项目的完整代码层次结构详解，请参见第 17.1 节。

```
e200_opensource
    |----rtl                                    // 存放 RTL 的目录
        |----e203                               // E203 核和 SoC 的 RTL 目录
            |----debug                          // 存放 Debug 相关模块的 RTL 代码
                |----sirv_debug_module.v        // 调试模块模块顶层
                |----sirv_debug_ram.v           // Debug-RAM 模块
                |----sirv_debug_rom.v           // Debug-ROM 模块
```

- 调试模块中实现了“0.11 版本”中定义的若干寄存器、Debug-ROM 和 Debug-RAM。这些资源既可以被调试总线访问到，也可以被系统存储总线访问到。有关此类寄存器、Debug-ROM、Debug-RAM 的细节和在不同总线上映射的地址区间请参见“0.11 版本”原文。调试模块的源代码片段如下所示。

```
// sirv_debug_module.v 源代码片段


// 系统存储总线 ICB 接口
```

```
    input                       i_icb_cmd_valid,
    output                      i_icb_cmd_ready,
    input  [12-1:0]             i_icb_cmd_addr,
    input                       i_icb_cmd_read,
    input  [32-1:0]             i_icb_cmd_wdata,

    output                      i_icb_rsp_valid,
    input                       i_icb_rsp_ready,
    output [32-1:0]             i_icb_rsp_rdata,

  ……

  // 解析来自DTM模块的调试总线

    assign dtm_req_bits_addr = i_dtm_req_bits[40:36];
    assign dtm_req_bits_data = i_dtm_req_bits[35:2];
    assign dtm_req_bits_op   = i_dtm_req_bits[1:0];
    assign i_dtm_resp_bits = {dtm_resp_bits_data, dtm_resp_bits_resp};
  ……

    // The OP field
    //   0: Ignore data. (nop)
    //   1: Read from address. (read)
    //   2: Read from address. Then write data to address. (write)
    //   3: Reserved.
    wire dtm_req_rd = (dtm_req_bits_op == 2'd1);
    wire dtm_req_wr = (dtm_req_bits_op == 2'd2);

    wire dtm_req_sel_dbgram   = (dtm_req_bits_addr[4:3] == 2'b0) & (~(dtm_req
_bits_addr[2:0] == 3'b111));//0x00-0x06
    wire dtm_req_sel_dmcontrl = (dtm_req_bits_addr == 5'h10);
    wire dtm_req_sel_dminfo   = (dtm_req_bits_addr == 5'h11);
    wire dtm_req_sel_haltstat = (dtm_req_bits_addr == 5'h1C);

  ……

  // 实现ICB总线读取Debug-ROM、Debug-RAM和寄存器

    assign i_icb_rsp_rdata =
              ({32{icb_sel_cleardebint}} & {{32-HART_ID_W{1'b0}}, cleardebint_r})
            | ({32{icb_sel_sethaltnot }} & {{32-HART_ID_W{1'b0}}, sethaltnot_r})
            | ({32{icb_sel_dbgrom   }} & rom_dout)
            | ({32{icb_sel_dbgram   }} & ram_dout);

  ……

  // 实现调试总线读取Debug-ROM、Debug-RAM和寄存器
    assign dtm_resp_bits_data =
              ({34{dtm_req_sel_dbgram  }} & {dmcontrol_r[33:32],ram_dout})
```

```
        | ({34{dtm_req_sel_dmcontrl}} & dmcontrol_r)
        | ({34{dtm_req_sel_dminfo   }} & dminfo_r)
        | ({34{dtm_req_sel_haltstat}} & {{34-HART_ID_W{1'b0}},dm_haltnot_r});
……
```

- Debug-ROM 模块中包含了处理器进入调试模式下需要执行的异常处理程序。该程序是“0.11 版本”中定义的固定程序，程序内容片段如图 14-3 所示，完整程序请参见“0.11 版本”完整文档。此段程序只读且不用更改，将其编译汇编成为最终的二进制代码之后，可以用一段 ROM 实现。相关源代码片段如下所示。

```
  // sirv_debug_rom.v 源代码片段

// These ROM contents support only RV32

// def xlen32OnlyRomContents : Array[Byte] = Array(
// 0x6f, 0x00, 0xc0, 0x03, 0x6f, 0x00, 0xc0, 0x00, 0x13, 0x04, 0xf0, 0xff,
// 0x6f, 0x00, 0x80, 0x00, 0x13, 0x04, 0x00, 0x00, 0x0f, 0x00, 0xf0, 0x0f,
// 0x83, 0x24, 0x80, 0x41, 0x23, 0x2c, 0x80, 0x40, 0x73, 0x24, 0x40, 0xf1,
// 0x23, 0x20, 0x80, 0x10, 0x73, 0x24, 0x00, 0x7b, 0x13, 0x74, 0x84, 0x00,
// 0x63, 0x1a, 0x04, 0x02, 0x73, 0x24, 0x20, 0x7b, 0x73, 0x00, 0x20, 0x7b,
// 0x73, 0x10, 0x24, 0x7b, 0x73, 0x24, 0x00, 0x7b, 0x13, 0x74, 0x04, 0x1c,
// 0x13, 0x04, 0x04, 0xf4, 0x63, 0x16, 0x04, 0x00, 0x23, 0x2c, 0x90, 0x40,
// 0x67, 0x00, 0x00, 0x40, 0x73, 0x24, 0x40, 0xf1, 0x23, 0x26, 0x80, 0x10,
// 0x73, 0x60, 0x04, 0x7b, 0x73, 0x24, 0x00, 0x7b, 0x13, 0x74, 0x04, 0x02,
// 0xe3, 0x0c, 0x04, 0xfe, 0x6f, 0xf0, 0x1f, 0xfe).map(_.toByte)

wire [31:0] debug_rom [0:28]; // 29 words in total

assign rom_dout = debug_rom[rom_addr];

 // 注意，代码中使用常数赋值实现，此模块如果直接使用综合工具综合，将会被优化为门数有限的组合逻辑。

// 0x6f, 0x00, 0xc0, 0x03, 0x6f, 0x00, 0xc0, 0x00, 0x13, 0x04, 0xf0, 0xff,
assign debug_rom[ 0][7 : 0] = 8'h6f;
assign debug_rom[ 0][15: 8] = 8'h00;
assign debug_rom[ 0][23:16] = 8'hc0;
assign debug_rom[ 0][31:24] = 8'h03;

assign debug_rom[ 1][7 : 0] = 8'h6f;
assign debug_rom[ 1][15: 8] = 8'h00;
assign debug_rom[ 1][23:16] = 8'hc0;
assign debug_rom[ 1][31:24] = 8'h00;

assign debug_rom[ 2][7 : 0] = 8'h13;
assign debug_rom[ 2][15: 8] = 8'h04;
assign debug_rom[ 2][23:16] = 8'hf0;
assign debug_rom[ 2][31:24] = 8'hff;
……
```

```
#include "riscv/encoding.h"

#define DEBUG_RAM          0x400
#define DEBUG_RAM_SIZE     64

#define CLEARDEBINT        0x100
#define SETHALTNOT         0x10c

        .global entry
        .global resume
        .global exception

        # Automatically called when Debug Mode is first entered.
entry:  j       _entry
        # Should be called by Debug RAM code that has finished execution and
        # wants to return to Debug Mode.
resume:
        j       _resume
exception:
        # Set the last word of Debug RAM to all ones, to indicate that we hit
        # an exception.
        li      s0, ~0
        j       _resume2

_resume:
        li      s0, 0
_resume2:
        fence
```

图 14-3 Debug-ROM 中的程序片段

- Debug-RAM 主要在运行 Debug-ROM 中固定异常处理程序时作为数据段使用，用于存放一些临时数据和中间数据。对于 32 位的 RISC-V 架构处理器而言，需要至少 28 个字节的数据空间，相关源代码片段如下所示。

```
// sirv_debug_ram.v 源代码片段


  wire [31:0] debug_ram_r [0:6];
  wire [6:0]  ram_wen;

  // 注意，代码中使用普通的寄存器实现了 7 个 32 比特宽的寄存器，而并非任何实际的 RAM。

  assign ram_dout = debug_ram_r[ram_addr];

  genvar i;
  generate //{

      for (i=0; i<7; i=i+1) begin:debug_ram_gen//{

            assign ram_wen[i] = ram_cs & (~ram_rd) & (ram_addr == i) ;
            sirv_gnrl_dfflr #(32) ram_dfflr (ram_wen[i], ram_wdat, debug_ra
m_r[i], clk, rst_n);

      end//}
  endgenerate//}
  ......
```

以上仅对最关键的代码片段予以讲解分析，完整源代码请读者到 GitHub 的 e200_opensource 项目中自行阅读。

14.3.4 调试中断处理

与第 13 章中描述的普通中断一样，调试中断作为一根输入信号输送给处理器的交付（Commit）模块，如图 13-10 中圆圈处所示。交付模块的调试中断处理部分的要点如下。

- 交付模块接受来自调试模块的一根中断信号的请求，由于中断是一种异步异常，这种中断异常的“发生指令 PC”将会由当前正在交付的指令承担，dpc 寄存器中更新的 PC 值即为当前正在交付指令（来自于 ALU 接口）。
- 调试中断一旦被接受，便会冲刷（Flush）流水线，将后续的指令取消掉，并向 IFU 模块发送冲刷请求（Flush Request）和重新取指的 PC（Flush PC），值为 0x800，用以重新从新的 PC 地址开始取指令。
- 交付模块中的调试中断处理相关源代码片段如下所示。

```
    // e203_exu_excp.v 源代码片段
......

  // 生成流水线冲刷请求，包括了调试中断造成的异常。

  assign excpirq_flush_req  = longp_excp_flush_req | dbg_entry_flush_req | ir
q_flush_req | alu_excp_flush_req;

  ......

  // 生成重新取指令的 PC（Flush PC），如果是调试中断造成的流水线冲刷，则会使用 0x800 作为新
的取指令 PC。
  assign excpirq_flush_pc = dbg_entry_flush_req ? 'E200_PC_SIZE'h800 : (all_e
xcp_flush_req & dbg_mode) ? 'E200_PC_SIZE'h808 : csr_mtvec_r;

  ......

  // 根据进入调试模式的触发条件，更新 CSR 寄存器 dcsr 中的 cause 域。

    wire [2:0] set_dcause_nxt =
                            dbg_trig_req ? 3'd2 :
                            dbg_ebrk_req ? 3'd1 :
                            dbg_irq_req  ? 3'd3 :
                            dbg_step_req ? 3'd4 :
                            dbg_halt_req ? 3'd5 :
                                         3'd0;
......
```

- e203_exu_excp 模块中的内容非常琐碎繁杂，必须了解 RISC-V 架构的很多细节才能理解。以上仅对关键的代码片段予以讲解分析，完整源代码请读者到 GitHub 的

e200_opensource 项目中自行阅读。

14.3.5 调试机制 CSR 寄存器的实现

如第 14.2.4 节中所述，“0.11 版本”中分别定义了若干 CSR 寄存器用于调试机制。相关源代码在 e200_opensource 目录的结构如下。

```
e200_opensource
    |----rtl                          // 存放 RTL 的目录
         |----e203                    // E203 核和 SoC 的 RTL 目录
              |----debug              // 存放调试相关模块的 RTL 代码
                   |----sirv_debug_csr.v    // 调试机制的 CSR 寄存器实现模块
```

在 sirv_debug_csr.v 中实现 CSR 寄存器基本上严格按照“0.11 版本”中的定义予以实现，请读者到 GitHub 的 e200_opensource 项目中自行阅读其源代码。

14.3.6 调试机制指令的实现

如第 14.2.3 节中所述，RISC-V 架构文档和“0.11 版本”中分别定义了 ebreak 和 dret 这两条用于调试机制的指令。

- ebreak 指令会触发处理器进入异常模式或者调试模式，其硬件实现与第 13 章中描述的其他异常一样。交付模块中的 ebreak 相关源代码片段如下所示。

```
  // e203_exu_excp.v 源代码片段

  // ebreak 指令由 ALU 执行，ALU 输出此指令的交付请求，交付模块根据当前的 dcsr 寄存器中的配置决定是跳入调试模式或是异常模式。

  // The ebreak instruction will generated regular exception when the ebreakm
  //    bit of DCSR reg is not set
  wire alu_excp_i_ebreak4excp =
                            alu_excp_i_ebreak
                          & ((~dbg_ebreakm_r) | dbg_mode);
  // The ebreak instruction will enter into the debug-mode when the ebreakm
  //    bit of DCSR reg is set
  wire alu_excp_i_ebreak4dbg = alu_excp_i_ebreak
                          & (~alu_need_flush)
                          & dbg_ebreakm_r
                          & (~dbg_mode);//Not in debug mode
......
```

- dret 指令会触发处理器退出调试模式。该指令在蜂鸟 E200 处理器中被当作一种跳转指令来执行，其硬件实现与第 9 章中描述的分支指令解析一样，在 e200_exu_branchslv 模块中处理，相关源代码片段如下所示。

```
    // e200_exu_branchslv.v 源代码片段

// 生成流水线冲刷请求，包括了 dret 指令。

  wire brchmis_need_flush = (
        (cmt_i_bjp & (cmt_i_bjp_prdt ^ cmt_i_bjp_rslv))
  //    If it is a FenceI instruction, it is always jump
        | cmt_i_fencei
  //    If it is a RET instruction, it is always jump
        | cmt_i_mret
  //    If it is a DRET instruction, it is always jump
        | cmt_i_dret
      );

// 如果是 dret 指令造成的冲刷，则会使用 dpc 的值作为重新取指令的 PC（Flush PC）。

    assign brchmis_flush_pc =
……
                              cmt_i_dret ? csr_dpc_r :
                              //cmt_i_mret ? csr_epc_r :
                                           csr_epc_r ;
……
```

14.3.7 小结

值得再次强调的是，调试系统的实现难度比处理器核更加复杂烦琐。在蜂鸟 E200 处理器的研发过程中，花费在调试系统上的时间远远超过处理器核本身，其实现细节丰富，本书只能予以简述。仅通过本章上述若干要点不足以完全理解 RISC-V 调试机制的硬件实现，因此作者强烈建议有兴趣的读者仔细研读“0.11 版本”的原文，结合蜂鸟 E200 开源的 Verilog 源代码加以研究，从而充分理解此部分内容。而对 Debug 系统软硬件实现细节无须深入了解的读者可以忽略此章，勿做深究。

第 15 章　动如脱兔，静若处子
——低功耗的诀窍



对于处理器而言，虽然我们非常关注其主频和性能，但有一个不可忽视的事实，那就是处理器在绝大多数的时间是处于待机休眠状态的。譬如我们日常使用的手机，绝大多数的时间是处于一种休眠状态。即便是在运行的过程中，大段的时间也是处于性能要求不高的场景。以知名的 ARM big-LITTLE 架构为例，是在使用能效比更高的小核运行于性能要求不高的场景，只有在最关键的时刻才启用耗电较高的大核。

所谓“动若脱兔，静若处子”，低功耗机制对于处理器而言至关重要。本章将对处理器的低功耗技术加以介绍，并结合蜂鸟 E200 处理器阐述其低功耗设计的诀窍。

15.1 处理器低功耗技术概述

处理器的低功耗技术可以从多个层面加以探讨，从高层的软件、系统，到底层的硬件工艺均可涉及。

15.1.1 软件层面低功耗

运行于处理器之上的是软件程序，是软件赋予了处理器灵魂。软件层面的灵活性很高，其发掘低功耗的效果比硬件低功耗本身的效果更加显著。通俗地讲，底层硬件辛辛苦苦地优化设计省的电，远远不如软件多休眠省的电多。

为了使处理器消耗尽可能少的功耗，一套好的软件程序应该尽可能合理地调用处理器的硬件资源，譬如以下情况。

- 仅在关键的场景调用耗能高的硬件，在一般的场景尽可能使用耗能低的硬件。
- 在处理器空闲的时刻，尽可能进入低功耗休眠模式，以节省功耗。

由于本书侧重于硬件设计，因此对软件层面的机制不做赘述。

15.1.2 系统层面低功耗

系统层面的低功耗技术可以涉及板级硬件系统和芯片内的 SoC 系统，其原理基本一致。以 SoC 系统为例，常见的低功耗技术如下。

- SoC 系统中划分不同的电源域，能够支持将 SoC 中的大部分硬件关闭电源。
- SoC 系统中划分不同的时钟域，能够支持小部分电路以低速低功耗的方式运行。
- 通过不同的电源域与时钟域的组合，划分出不同的低功耗模式。SoC 配备 PMU（Power Management Unit）控制进入或者退出不同的低功耗模式。
- 软件可以通过使用 PMU 的功能，在不同的场景下进入和退出不同的低功耗模式。

第 15.3 节将以蜂鸟 E200 SoC 系统为例阐述上述宗旨。

15.1.3 处理器层面低功耗

处理器层面的常见的低功耗技术如下。

（1）处理器指令集中定义一种休眠指令，运行该指令后处理器核便进入休眠状态。

（2）休眠状态可分为浅度休眠和深度休眠。

- 浅度休眠状态往往将处理器核的整个时钟关闭，但仍然保留电源供电，因此可以节省动态功耗，但是静态漏电功耗仍然有消耗。
- 深度休眠状态不仅关闭处理器核的时钟，甚至将电源也关闭，因此可以同时省掉动态和静态功耗。

（3）处理器核深度休眠断电后，其内部上下文状态可以有两种策略进行保存和恢复。

- 策略一：在处理器核内部使用具有低功耗维持（Retention）能力的寄存器或者 SRAM 保存处理器状态，这种寄存器或者 SRAM 在主电源被关闭后可以使用极低的漏电消耗保存处理器的状态。
- 策略二：使用软件的保存恢复（Save-and-Restore）机制，即在断电前将处理器的上下文状态保存在 SoC 层面的电源常开域（Power Always-on Domain）中，待到唤醒恢复供电后，使用软件从电源常开域中读取回来加以恢复。

策略一的优点是休眠和唤醒的速度极快，但是 ASIC 设计的复杂度高；策略二的优点是实现非常简单，但是休眠和唤醒的速度相对较慢。

（4）在处理器的架构上，可以采用异构的方式节省功耗。

- 有关异构的典型示例，请参见第 16.1 节了解更多细节。
- 另一种知名度很高的示例便是 ARM big-LITTLE 架构，其使用能效比更高的小核运行于性能要求不高的场景，只有在最关键的时刻才启用耗电较高的大核，从而节省动态功耗。

15.1.4 单元层面低功耗

模块和单元层面的低功耗技术已经进入了 IC 设计微架构的范畴。其常见的技术与 SoC 系统层面基本一致，只不过是规模更小的版本。

- 一个功能完整的单元往往需要单独配备独立的时钟门控（Clock Gate），当该模块或者单元空闲时，可以使用时钟门控将其时钟关闭以节省动态功耗。
- 某些比较独立和规模较大的模块甚至可以划分独立的电源域来支持关闭电源，以进一步节省静态功耗。

第 15.3 节将以蜂鸟 E200 处理器核为例来阐述上述宗旨。

15.1.5 寄存器层面低功耗

寄存器层面的低功耗技术已经进入了 IC 设计编码风格的范畴，可以从以下 3 个方面减少寄存器层面的功耗。

（1）时钟门控。

- 目前主流的逻辑综合工具均有从代码风格中直接推断出 ICG（Integrated Clock Gating）的能力。因此只要遵循一定的编码风格，便能够将一组寄存器的时钟自动推断出 ICG，以节省动态功耗。
- 在逻辑综合完成后，工具可以生成整个电路的“时钟门控率（Clock Gating Rate）”。开发者可以通过此“时钟门控率”数据的高低，来判断其设计的电路是否被自动推断出了足够的 ICG。好的电路一般有超过 90%的“时钟门控率”，否则可能是电路中数据通路较少（主要以小位宽寄存器为基础的控制电路为主），或者编码风格有问题。

（2）减少数据通路翻转。

为了减少不必要的动态功耗，应该尽量减少寄存器的翻转。

- 示例一：以处理器的流水线为例，每级流水线通常需要配置一位控制位（Valid 位）表示该级流水线是否有有效指令。当指令加载至此级流水线时将 Valid 位设为高，离开此级流水线时将 Valid 位清零。但是对于此级流水线的数据通路载体部分（Payload 部分），只有在指令加载至此级流水线时，向载体（Payload）部分的寄存器加载指令信息（通常有数十位），而指令离开此级流水线时，载体部分的寄存器无须清零。通过此方法能够极大减少数据通路部分的寄存器翻转率。
- 示例二：以 FIFO（当容量较小而使用寄存器作为存储部分）设计为例，虽然理论上可以使用比较简单的数据表项逐次移位的方式，实现 FIFO 的先入先出功能，但是却应该使用维护读写指针的方式（数据表项寄存器则不用移位）实现先入先出的功能。因为数据表项逐次移位的方式会造成寄存器的大量翻转，相比而言，使用读写指针的方式实现则保持了表项寄存器中的值静止不动，从而大幅减少动态功耗，因此应该优先采用此方法。

（3）数据通路不复位。

- 与上一点同理，对于数据通路部分的寄存器，甚至可以使用不带复位信号的寄存器。不带复位信号的寄存器面积更小，时序更优，功耗更低。譬如对于某些缓冲器（Buffer）、FIFO 和 Regfile 的寄存器部分，经常使用不带复位的寄存器。
- 但使用不带复位的寄存器时必须小心谨慎，保证其没有作为任何其他控制信号，以免造成不定态的传播。在前仿真阶段，必须有完善的不定态捕捉机制发现这些问题，否则可能造成芯片的严重 Bug。蜂鸟 E200 的设计编码风格便能够提供强大的不定态

捕捉机制，请参见第 5.3 节了解有关信息。

15.1.6 锁存器层面低功耗

锁存器相比寄存器面积更小，功耗更低。在某些特定的场合使用可以降低芯片功耗，但是锁存器会给数字 ASIC 流程带来极大困扰，因此应该谨慎使用。

15.1.7 SRAM 层面低功耗

SRAM 在芯片设计中经常使用到，可以从以下 3 个方面减少 SRAM 的功耗。

（1）选择合适的 SRAM。

- 常规 SRAM 通常分为“单口 SRAM（Single Port SRAM）”“一读一写 SRAM（Two-Port Regfile）”“双口 SRAM（Dual-Port SRAM）”。其他类型的 SRAM 需要特殊定制。
- 从功耗与面积的角度来讲，单口 SRAM 最小，一读一写 Regfile 其次，双口 SRAM 最大。应该优先选择功耗与面积小者，尽量避免使用高功耗的 SRAM 类型。
- SRAM 的数据宽度也会影响其面积。以同等大小的 SRAM 为例，假设总容量为 16KB，如果 SRAM 的数据宽度为 32 位，则深度为 4096。如果 SRAM 的数据宽度为 64 位，则深度为 2048。不同的宽度深度比可能会产生面积迥异的 SRAM，因此也需要综合权衡。

（2）尽量减少 SRAM 读写。

- SRAM 的读写动态功耗相当可观，因此应该尽量减少读写 SRAM。
- 以处理器取指令为例，由于处理器多数按顺序取指，因此应该尽量一次从 SRAM 中多读回一些指令，而不是反复多次地读取 SRAM（一次读一点点指令），从而节省 SRAM 的动态功耗。

（3）空闲时关闭 SRAM。

- 与单元门控时钟相同的原理，在空闲时应关闭 SRAM 的时钟，以节省动态功耗。
- SRAM 的漏电功耗相当可观，因此在省电模式下，可以将 SRAM 的电源关闭，以防止漏电。

15.1.8 组合逻辑层面低功耗

组合逻辑是芯片中的基本逻辑，可以从以下两个方面减少组合逻辑的功耗。

（1）减少面积。

通过使用尽量少的组合逻辑面积减少静态功耗，此为数字逻辑设计的基本认知，无须赘述。因此从设计思路和代码风格上，应该尽量将大的数据通路（或者运算单元）进行复用，从而减少面积。另外应该避免使用除法、乘法等大面积的运算单元，尽量将其转化为加减法运算。

（2）减少翻转率。

可以通过逻辑门控的方式，在数据通路上加入一级“与”门，使没有用到的组合逻辑在空闲时不翻转，从而达到减少动态功耗的效果。额外加入一级与门，在时序非常紧张的场合也许无法接受，需要谨慎使用。

15.1.9 工艺层面低功耗

工艺层面的低功耗一般涉及使用特殊的工艺单元库，本书在此不做过多探讨。

15.2 RISC-V 架构的低功耗机制

处理器指令架构本身并不会定义低功耗机制，但是如第 15.1.3 节中所述，处理器架构中通常会定义一条休眠指令，本节将介绍 RISC-V 架构定义的 WFI 指令。

WFI 指令

WFI（Wait For Interrupt）指令是 RISC-V 架构定义的专门用于休眠的指令。当处理器执行到 WFI 指令之后，将会停止执行当前的指令流，进入一种空闲状态。这种空闲状态可以被称为“休眠”状态，直到处理器接收到中断（中断局部开关必须被打开，由 mie 寄存器控制），处理器便被唤醒。处理器被唤醒后，如果中断被全局打开（mstatus 寄存器的 MIE 域控制），则进入中断异常服务程序开始执行；如果中断被全局关闭，则继续顺序执行之前停止的指令流。

以上是 RISC-V 架构推荐的行为，在具体的硬件实现中，WFI 指令也可以被当成一种 NOP 操作，即什么也不干（并不真正支持休眠模式）。关于 WFI 指令的更多细节，请参阅附录 A14.2 节。

15.3 蜂鸟 E200 低功耗机制的硬件实现

软件层面低功耗机制超出了本书讨论的范畴，在此不做赘述。本节将从系统、处理器、单元、寄存器、锁存器、SRAM、组合逻辑以及工艺层面阐述蜂鸟 E200 处理器的低功耗机制。

15.3.1 蜂鸟 E200 系统层面低功耗

蜂鸟 E200 配套的 SoC 结构如图 15-1 所示。请参见第 18.2 节了解更多蜂鸟 E200 的 SoC 细节。

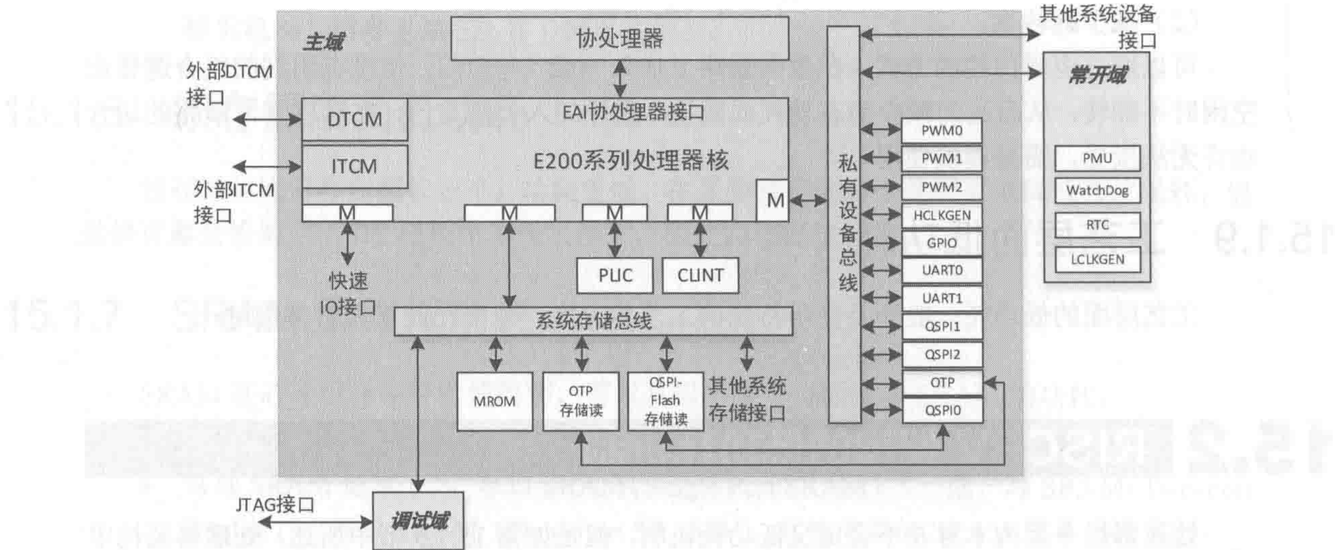


图 15-1　蜂鸟 E200 配套 SoC 总体结构图

蜂鸟 E200 配套的 SoC 整体分为 3 个时钟域（Clock Domains），如表 15-1 所示。

表 15-1　　蜂鸟 E200 配套 SoC 的时钟域

时钟域	介绍
JTAG 时钟域 (JTAG Clock Domain)	JTAG 接口的相关逻辑使用 JTAG 时钟
主域 (Main Domain)	由于蜂鸟 E200 处理器核的主频比较低，该 SoC 中对时钟域的划分相对比较简单，将所有的外设、存储和总线以及处理器核均放置于一个时钟域，用户可以自行修改将总线或者外设放于不同的时钟域
常开域 (Always-on Domain)	此域使用极其低速的时钟，因为此域中主要包含看门狗计数器（Watch Dog Timer）、实时计数器（Real-Time Counter）这种永不停歇的计时器模块。如果使用高速时钟不断计数，会造成大量的功耗损失，因此必须使用低速的时钟作为时钟频率控制计时器计数 请参见第 18.2.1 节了解更多常开域的细节

蜂鸟 E200 配套的 SoC 整体可以分为 3 个电源域（Power Domains），如表 15-2 所示。

表 15-2　　蜂鸟 E200 配套 SoC 的电源域

电源域	介绍
调试域 (Debug Domain)	此域中包含第 14 章中描述的所有调试相关的硬件模块。对于那些不需要调试功能的场景，可以选择将此域关电以节省功耗
主域 (Main Domain)	该 SoC 中对于电源域的划分相对比较简单，将所有的外设、存储和总线以及处理器核均放置于一个电源域，用户可以自行修改将总线或者外设放于不同的电源域
常开域 (Always-on Domain)	此域中主要包含看门狗计数器、实时计数器这种永不停歇的计时器模块。另外，此域中还包含一个电源管理单元（Power Management Unit），可以用来控制其他电源域的开和关

通过合理地关闭不同的电源域，便可以进入不同的低功耗模式。譬如，软件可以将整个主域和调试域电源关闭，仅保留常开域的电源。通过配置 PMU 使用实时计数器的中断作为唤醒条件，将整个系统重新唤醒。请参见第 18.2.1 节了解更多 PMU 的细节。

注意：在 GitHub 网站上 e200_opensource 项目的源代码中并没有实现任何电源域相关的逻辑，多电源域的设计目前需要特定 ASIC 工艺和流程的支持，请读者自行实现。

15.3.2 蜂鸟 E200 处理器层面低功耗

蜂鸟 E200 处理器层面的低功耗主要在于对 WFI 指令的实现，硬件实现要点如下。

- e203_cpu_top 顶层模块有一个输出信号 core_wfi。当该信号为高时，表示处理器核已经进入了休眠模式。系统 SoC 可以通过检测此输出信号确定处理器已经进入休眠状态，继而安全地关闭其电源。
- 蜂鸟 E200 处理器核在执行了 WFI 指令之后将阻止处理器执行后续的指令，并要求处理器核中所有的单元完成正在执行的操作（譬如完成已经发出的总线操作）。待到满足条件后，便意味着可以安全地进入休眠模式，将输出信号 core_wfi 置高。在进入休眠模式后，如果有新的中断到来，则会重新唤醒处理器，并将输出信号 core_wfi 置低。相关模块的源代码在 e200_opensource 目录的结构和源代码片段如下。

```
e200_opensource
    |----rtl                                        // 存放 RTL 的目录
        |----e203                                   // E203 核和 SoC 的 RTL 目录
            |----core                               // 存放处理器核相关模块的 RTL 代码
                |----e203_exu_disp.v           // 指令派遣模块
                |----e203_exu_excp.v           // 中断和异常处理模块
```

```
   // e203_exu_disp.v 源代码片段

  // 如果已经执行了 WFI 指令，派遣模块便会接收到来自交付模块要求 EXU 单元完成所有操作并准备休眠的请求信号 wfi_halt_exu_req，以阻止其派遣后续的指令。

    wire disp_condition =
  ......
                  // If it was a WFI instruction commited halt req, then it will stall the disaptch
                  & (~wfi_halt_exu_req)

  // 等待所有已经滞外的指令执行完毕（OITF 变空），作为表征 EXU 单元已经完成所有操作可以进入休眠的反馈信号。
    assign wfi_halt_exu_ack = oitf_empty;
```

```
    // e203_exu_excp.v 源代码片段

    wire wfi_req_hsked = wfi_halt_ifu_req & wfi_halt_ifu_ack & wfi_halt_exu_r
eq & wfi_halt_exu_ack;

    // core_wfi 信号在执行了 WFI 指令并且其他单元都已经完成了所有正在执行的操作后，将被置高。

    wire wfi_flag_set = wfi_req_hsked;

    // core_wfi 信号在收到了新的中断请求后，或者进入调试模式的请求后，将被置低。

    wire wfi_irq_req;
    wire dbg_entry_req;
    wire wfi_flag_r;
    wire wfi_flag_clr = wfi_irq_req | dbg_entry_req;
    wire wfi_flag_ena = wfi_flag_set | wfi_flag_clr;
      // If meanwhile set and clear, then clear preempt
    wire wfi_flag_nxt = wfi_flag_set & (~wfi_flag_clr);
    sirv_gnrl_dfflr #(1) wfi_flag_dfflr (wfi_flag_ena, wfi_flag_nxt, wfi_flag
_r, clk, rst_n);
    assign core_wfi = wfi_flag_r & (~wfi_flag_clr);
```

蜂鸟 E200 处理器核在顶层配备了专门的时钟控制模块，用于控制处理器核的时钟关闭。时钟控制模块的源代码在 e200_opensource 目录的结构如下。

```
e200_opensource
    |----rtl                                    // 存放 RTL 的目录
         |----e203                              // E203 核和 SoC 的 RTL 目录
              |----core                         // 存放处理器核相关模块的 RTL 代码
                   |----e203_clk_ctrl.v         // 时钟控制模块
```

当处理器核执行了 WFI 指令之后，时钟控制模块将处理器核中所有单元的时钟门控均关闭，从而进入休眠状态。相关源代码片段如下所示。

```
    // e203_clk_ctrl.v 源代码片段

  // 使用 core_wfi 信号强行将时钟门控的使能信号置低。

      // The IFU is always actively fetching unless there is WFI to override it
    wire ifu_clk_en = (core_ifu_active & (~core_wfi));

  // 时钟门控的使能信号用于门控时钟的生成
    e203_clkgate u_ifu_clkgate(
      .clk_in   (clk          ),
      .test_mode(test_mode    ),
      .clock_en (ifu_clk_en),
      .clk_out  (clk_core_ifu) // 用于 IFU 单元的时钟
    );
```

15.3.3 蜂鸟 E200 单元层面低功耗

蜂鸟 E200 处理器核的几个主要的功能单元均配备了独立的时钟门控，一旦单元处于空闲的周期，即自动地将时钟关闭，从而节省动态功耗。典型的源代码片段如下所示。

```
    // e203_clk_ctrl.v 源代码片段


  // core_lsu_active 信号表征 LSU 单元目前是否空闲，如果该信号为低电平，则意味着空闲，将
lsu_clk_en 信号置低。

   wire lsu_clk_en = core_lsu_active;

  // 如果 lsu_clk_en 信号为低，则将门控时钟关闭
   e203_clkgate u_lsu_clkgate(
     .clk_in    (clk        ),
     .test_mode(test_mode  ),
     .clock_en  (lsu_clk_en),
     .clk_out   (clk_core_lsu)
   );
```

15.3.4 蜂鸟 E200 寄存器层面低功耗

如第 15.1.5 节所述，寄存器层面低功耗可以从“时钟门控”“减少数据通路翻转”“数据通路不复位”3 个方面减少功耗。下面以蜂鸟 E200 的源代码为例，分别予以阐述。

1. 时钟门控

蜂鸟 E200 遵循严格的代码风格，将所有的寄存器编码为模块化的 D 触发器模块（DFF-Module），从而方便综合工具轻松地识别其 Load-Enable 信号，继而推断出 ICG，取得很高的时钟门控率。请参见第 5.3 节了解更多有关蜂鸟 E200 处理器核 RTL 代码风格的信息。

模块化的 DFF-Module 的源代码在 e200_opensource 目录的结构如下。

```
e200_opensource
     |----rtl                               // 存放 RTL 的目录
          |----e203                         // E203 核和 SoC 的 RTL 目录
               |----general                 // 存放一些公用的通用 RTL 代码
                    |----sirv_gnrl_dffs.v       //模块化的 DFF-Modules
```

典型的源代码片段如下所示。

```
    // e203_grnl_dffs.v 源代码片段


  // 该模块被调用例化生成带有 Load-Enable，异步 Reset 的 D 触发器（D Flip-Flops）
```

```
module sirv_gnrl_dfflrs # (
  parameter DW = 32
) (

  input               lden,
  input     [DW-1:0] dnxt,
  output    [DW-1:0] qout,

  input               clk,
  input               rst_n
);

reg [DW-1:0] qout_r;

always @(posedge clk or negedge rst_n)
begin : DFFLRS_PROC
  if (rst_n == 1'b0)
    qout_r <= {DW{1'b1}};
  else if (lden == 1'b1)  // 明确的 Load-Enable 信号便于综合工具轻松地识别推断出 ICG
    qout_r <= dnxt;
end

assign qout = qout_r;

endmodule
```

2. 减少数据通路翻转

蜂鸟 E200 遵循如第 15.1.5 节中所述的原则，流水线或者数据通路的 Payload 部分只有在流水线加载时更新。在流水线清空之时，寄存器中的值并不会清除，从而减少数据通路的寄存器翻转。

典型的一级流水线模块的源代码在 e200_opensource 目录的结构如下。

```
e200_opensource
    |----rtl                        // 存放 RTL 的目录
         |----e203                  // E203 核和 SoC 的 RTL 目录
              |----general          // 存放一些公用的通用 RTL 代码
                   |----sirv_gnrl_bufs.v     //存放一级流水线模块的源代码
```

一级流水线模块的源代码片段如下所示。

```
    // sirv_gnrl_pipe_stage.v 源代码片段

// 流水线会配备一个有效控制位（valid）寄存器

      wire vld_set;
      wire vld_clr;
      wire vld_ena;
      wire vld_r;
      wire vld_nxt;
```

```
    // 有效位寄存器在流水线加载时置高
    assign vld_set = i_vld & i_rdy;
    // 有效位寄存器在流水线清空时清零
    assign vld_clr = o_vld & o_rdy;

    assign vld_ena = vld_set | vld_clr;     //有效位寄存器在加载或者清空时使能
    assign vld_nxt = vld_set | (~vld_clr); // 置高或者清零，若同时发生，置高优先

      //例化有效控制位寄存器
    sirv_gnrl_dfflr #(1) vld_dfflr (vld_ena, vld_nxt, vld_r, clk, rst_n);

//Payload 部分的数据通路只有在流水线加载时使能翻转，因此其 Load-enable 使用 vld_set 信号
    sirv_gnrl_dffl #(DW) dat_dfflr (vld_set, i_dat, o_dat, clk);
```

3. 数据通路不复位

蜂鸟 E200 遵循如第 15.1.5 节所述的原则，对于大片的纯数据通路（非控制信号）寄存器不使用复位信号，从而减少面积与功耗。典型的模块，如 FIFO（使用寄存器作为存储部分）模块和通用寄存器组（Regfile）模块。其源代码在 e200_opensource 目录的结构如下。

```
e200_opensource
     |----rtl                            // 存放 RTL 的目录
          |----e203                      // E203 核和 SoC 的 RTL 目录
               |----general              // 存放一些公用的通用 RTL 代码
                    |----sirv_gnrl_bufs.v     //存放 FIFO 模块的源代码
               |----core
                    |----e203_exu_regfile.v  //存放 Regfile 模块的源代码
```

FIFO 模块的源代码片段如下所示。

```
    // sirv_gnrl_fifo.v 源代码片段

  for (i=0; i<DP; i=i+1) begin:fifo_rf//{
      assign fifo_rf_en[i] = wen & wptr_vec_r[i];
      // FIFO 的存储寄存器部分不使用 Reset 复位信号
  sirv_gnrl_dffl  #(DW) fifo_rf_dffl (fifo_rf_en[i], i_dat, fifo_rf_r[i], clk);
end//}
```

Regfile 模块的源代码片段如下所示。

```
    // e203_exu_regfile.v 源代码片段

  generate //{

      for (i=0; i<'E203_RFREG_NUM; i=i+1) begin:regfile//{
......
        else begin: rfno0
            assign rf_wen[i] = wbck_dest_wen & (wbck_dest_idx == i) ;
          'ifdef E203_REGFILE_LATCH_BASED //{
            e203_clkgate u_e203_clkgate(
              .clk_in  (clk  ),
              .test_mode(test_mode),
```

```
                .clock_en(rf_wen[i]),
                .clk_out (clk_rf_ltch[i])
              );
                 //from write-enable to clk_rf_ltch to rf_ltch
              sirv_gnrl_ltch #('E203_XLEN) rf_ltch (clk_rf_ltch[i], wbck_dest
_dat_r, rf_r[i]);
            'else//}{
             //如果使用寄存器实现的 Regfile，其寄存器不使用 Reset 复位信号
              sirv_gnrl_dffl #('E203_XLEN) rf_dffl (rf_wen[i], wbck_dest_dat,
 rf_r[i], clk);
            'endif//}
          end

      end//}
  endgenerate//}
```

15.3.5 蜂鸟 E200 锁存器层面低功耗

如第 15.1.6 节所述，锁存器相比寄存器面积更小，功耗更低。在某些特定的场合使用可以降低芯片功耗。在蜂鸟 E200 的实现中，通用寄存器组（Regfile）模块可以配置为基于锁存器的实现，从而大幅减少 Regfile 的面积。

注意：锁存器会给数字 ASIC 流程带来极大困扰，因此应该谨慎使用此配置。Regfile 模块的源代码片段如下所示。

```
      // e203_exu_regfile.v 源代码片段

  generate //{

      for (i=0; i<'E203_RFREG_NUM; i=i+1) begin:regfile//{
......
        else begin: rfno0
              assign rf_wen[i] = wbck_dest_wen & (wbck_dest_idx == i) ;
            'ifdef E203_REGFILE_LATCH_BASED //{
              e203_clkgate u_e203_clkgate(
                .clk_in  (clk  ),
                .test_mode(test_mode),
                .clock_en(rf_wen[i]),
                .clk_out (clk_rf_ltch[i])
              );
                 //可以配置使用锁存器实现 Regfile
              sirv_gnrl_ltch #('E203_XLEN) rf_ltch (clk_rf_ltch[i], wbck_dest
_dat_r, rf_r[i]);
            'else//}{
        sirv_gnrl_dffl #('E203_XLEN) rf_dffl (rf_wen[i], wbck_dest_dat, rf_
r[i], clk);
            'endif//}
          end

      end//}
  endgenerate//}
```

15.3.6 蜂鸟 E200 SRAM 层面低功耗

如第 15.1.7 节所述，SRAM 层面低功耗可以从“选择合适的 SRAM”“尽量减少 SRAM 读写”“空闲时关闭 SRAM”3 个方面减少功耗。以下以蜂鸟 E200 源代码为例，分别予以阐述。

（1）选择合适的 SRAM。

- 蜂鸟 E200 的 ITCM 和 DTCM 均需使用 SRAM。如第 15.1.7 节所述，单口 SRAM 在 3 种不同的 SRAM 类型中最为省电，因此为了减少功耗和面积，蜂鸟 E200 均采用单口 SRAM 实现 ITCM 和 DTCM。
- 如第 15.1.7 节所述，SRAM 的形状也能影响功耗的大小。蜂鸟 E200 的 ITCM SRAM 宽度选择为 64 位，之所以选择 64 位宽，是因为对于同等容量的 SRAM 而言，64 位宽的 SRAM 比 32 位宽的 SRAM 具有更好的面积压缩比，从而减少功耗开销。

（2）尽量减少 SRAM 读写。

- 如第 15.1.7 节所述，尽量减少 SRAM 的读写能够有效降低功耗。蜂鸟 E200 的 ITCM SRAM 宽度选择为 64 位，同样可以减少 ITCM SRAM 的读功耗。这是因为处理器在取指令时，多数情况为按顺序取指，因此 64 位宽的 ITCM 可以一次取出 64 位的指令流，相比于从 32 位宽的 ITCM 中需要连续读两次才取出 64 位的指令流，只读一次 64 位宽的 SRAM 能够消耗更少的动态功耗。
- 由于蜂鸟 E200 ITCM 的 SRAM 宽度为 64 位，因此其输出为一个与 64 位地址区间对齐的数据，在此称之为一个“Lane”。假设是地址自增的顺序取指，由于 IFU 每次只取 32 位，因此会连续两次或者多次在同一个 Lane 里面访问。如果上次已经访问了 ITCM 的 SRAM，则下一次取指在同一个 Lane 的访问不会再次真的读 SRAM（不会打开 SRAM 的 CS 使能），而是利用 SRAM 输出保持不变的特点，直接使用其保持不变的输出，这样可以省却 SRAM 重复打开造成的动态功耗。请参见第 7.3.5 节了解有关 ITCM SRAM 实现的更多详情。
- 此外，蜂鸟 E200 ITCM 的 SRAM 宽度为 64 位相对于 32 位的 SRAM 而言，能够进一步减少取指令落入地址非对齐边界的概率（如果 SRAM 为 32 位宽则较多概率落入 32 位非对齐的地址边界，而 64 位宽的 SRAM 仅在 64 位的地址边界发生非对齐），从而减少非对齐取指令造成的性能和功耗损失。请参见第 7.3.5 节了解有关非对齐取指令实现的更多详情。

（3）空闲时关闭 SRAM。

蜂鸟 E200 的 SRAM 均配备独立的门控时钟单元，以减少动态功耗，典型代码片段如下所示。

```
// sirv_1cyc_sram_ctrl.v 源代码片段

// 此模块被例化于 e203_itcm_ctrl 与 e203_dtcm_ctrl 模块中用于控制 ITCM 和 DTCM 的 SRAM 模块读写

  assign ram_cs = uop_cmd_valid & uop_cmd_ready;
  assign ram_we = (~uop_cmd_read);
  assign ram_addr= uop_cmd_addr [AW-1:AW_LSB];
  assign ram_wem = uop_cmd_wmask[MW-1:0];
  assign ram_din = uop_cmd_wdata[DW-1:0];

  wire ram_clk_en = ram_cs;

// 为 SRAM 配备独立的时钟门控单元，只有在访问 SRAM 时（CS 为高）才将其时钟打开。
  e203_clkgate u_ram_clkgate(
    .clk_in    (clk        ),
    .test_mode(test_mode   ),
    .clock_en (ram_clk_en),
    .clk_out   (clk_ram)
  );

  assign uop_rsp_rdata = ram_dout;
```

15.3.7 蜂鸟 E200 组合逻辑层面低功耗

如第 15.1.8 节所述，组合逻辑层面低功耗可以从“减少面积”和“减少翻转率”两个方面减少功耗。以下以蜂鸟 E200 源代码为例，分别予以阐述。

（1）减少面积。

蜂鸟 E200 设计的一个重要目标便是尽量减少面积实现超低功耗，因此从设计思路和代码风格上尽量将大的数据通路（或者运算单元）进行复用，从而减少面积，情况如下所示。

- ALU 中的数据通路被充分复用（参见第 8.3.8 节了解 ALU 的实现细节）、多周期乘除法器也共用数据通路（参见第 8.3.9 节了解整数乘除法器的实现细节）。
- 在蜂鸟 E200 的源代码设计之中，无处不在地尽量进行着资源复用，本书在此不一一赘述，感兴趣的读者可以在阅读源代码时自行体会。

（2）减少动态功耗。

蜂鸟 E200 设计的另外一个重要目标便是尽量降低翻转率，以实现超低功耗，因此从设计思路和代码风格上尽量减少了组合逻辑的翻转率，甚至在某些情况下牺牲了时序。

- 蜂鸟 E200 的每个运算单元的输入信号均额外加入了一级“与”门，当每个运算单元不被使用时，其输入信号被“与”门屏蔽成为 0，从而使运算单元的输入组合逻辑部分在空闲时不发生翻转，减少动态功耗。
- 由于蜂鸟 E200 寄存器组（Regfile）模块的每个读端口都是一个纯粹的并行多路选择

器，多路选择器的选择信号为读操作数的寄存器索引。为了减少功耗，读端口的寄存器索引信号被专用的寄存器进行寄存，只有在执行需要读操作数的指令时才会被加载（否则保持不变），从而减少读端口的动态翻转功耗。

15.3.8 蜂鸟 E200 工艺层面低功耗

工艺层面的低功耗一般涉及使用特殊的工艺单元库，本书在此不做过多探讨。

15.4 总结

蜂鸟 E200 处理器核虽然是一款开源处理器核，但是蜂鸟 E200 系列处理器研发团队拥有多年在国际一流公司开发处理器的经验，使用严格的工业界标准进行设计和编码。如本章各节中所述，蜂鸟 E200 处理器核从各个层面使用严谨的方法进行低功耗设计，不逊色于任何其他商用的处理器核 IP。

第 16 章　工欲善其事，必先利其器
——RISC-V 可扩展协处理器



本章将介绍如何利用 RISC-V 的可扩展性，并以蜂鸟 E200 的协处理器接口为例详细阐述如何定制一款协处理器。

16.1 专用领域架构 DSA

熟悉计算机体系结构的读者可能熟知“异构计算”的概念，异构计算的直接解释是指不同指令集架构的几种处理器组合在一起进行计算。异构计算的精髓并不在于异构本身，其核心的理念在于使用专业的硬件做专业的事情，典型的例子是 CPU+GPU 的组合，CPU 侧重于通用的控制和计算，而 GPU 则侧重于专用的图像处理。研究表明多核异构计算由于利用了专业的特性，可以获得比普通同构架构更高的性能，而消耗更少的功耗。

与异构计算原理相同而更加通俗、亲民的另外一个概念便是专用领域架构（Domain Specific Architecture，DSA）。著名的计算机体系结构领域泰斗 John Hennessy 教授在 2017 年的演讲一次中提到，目前处理器发展的新希望在于 DSA。John Hennessy 教授将 1977～2017 年间的 40 年称之为处理器发展的“黄金 40 年”。在这 40 年间，处理器以令人惊异的速度发展，处理器的性能以平均每年 1.4 倍的速度呈指数提高，相比最早期的处理器，当今处理器达到了上百万倍的性能提升。随着摩尔定律的发展逼近极限，处理器架构的发展也遭遇了瓶颈。单核指令级并行度从早期需要平均 4～10 个时钟周期完成一条指令到如今一个周期可以执行超过 4 条指令；时钟频率从早期 3MHz 发展到如今 4GHz；处理器核数从早期的单核发展到数十个核。这三个方面的发展目前均已逼近极限，同时处理器的应用领域也发展到了更加多样的云端、移动端、深嵌入式端等领域，且能效比正成为最重要的指标。譬如，处理器在移动设备中已经成为继屏幕之后能量消耗最大的元件，因此移动设备中处理器能效比是至关重要的问题。而在另一个未来的处理器大型市场——云端服务器市场，能效比也是十分关键的指标。在数据中心的成本中，散热已经成了最高的成本之一。为了减少成本必须考虑处理器能效比，处理器架构必须改善能效比，但是传统通用架构设计方法的能效比已经到了极限。

为了进一步提高能效比，John Hennessy 教授指出，处理器架构的希望在于专用领域处理器架构。DSA 的核心思想同样是使用专用的硬件做专用的事情，但是与 ASIC 硬件化的电路不同，DSA 是满足一个 Domain 内的应用，而非一个固定的应用，因此它能够满足灵活性与专用性的折衷。同时它需要更多专用领域的专业知识，从而更好地为 Domain Specific 设计出更合适的架构。

DSA 有时也被解释为 Domain Specific Accelerator，即对主处理器适当地扩展出面向某些特定领域的协处理器加速器，这种“Domain Specific Accelerator”也是“Domain Specific Architecture”的体现，能极大地提高能效比。

16.2 RISC-V 架构的可扩展性

RISC-V 架构的显著特性之一便是开放的可扩展性，从而能够非常容易在 RISC-V 的通用架构基础上实现 Domain Specific Accelerator，这也是 RISC-V 架构相比 ARM 和 x86 等主流商业架构的最大优点。RISC-V 的可扩展性体现在如下两个方面：

- 预留的指令编码空间。
- 预定义的 Custom 指令。

16.2.1 RISC-V 的预留指令编码空间

RISC-V 架构定义的标准指令集仅使用了少部分的指令编码空间，更多的指令编码空间被预留给用户作为扩展指令使用。由于 RISC-V 架构支持多种不同的指令长度，不同的指令长度均预留有不同的编码空间。以最常用的 32 位和 16 位长度指令为例，如表 16-1 和表 16-2 所示，指令的低 7 位为 opcode，各种不同的 opcode 值的组合代表了不同的指令类型，譬如 AMO 指令和 OP-FP（浮点）指令。用户可以从 3 个方面利用 RISC-V 预留的编码空间。

表 16-1 RISC-V 架构 32 位指令 opcode 表（inst[1:0]==11）

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (>32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥80*b*

表 16-2 RISC-V 架构 16 位指令 opcode 表

inst[15:13] inst[1:0]	000	001	010	011	100	101	110	111	
00	ADDI4SPN	FLD FLD LQ	LW	FLW LD LD	*Reserved*	FSD FSD SQ	SW	FSW SD SD	RV32 RV64 RV128
01	ADDI	JAL ADDIW ADDIW	LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64 RV128
10	SLLI	FLDSP FLDSP LQ	LWSP	FLWSP LDSP LDSP	J[AL]R/MV/ADD	FSDSP FSDSP SQ	SWSP	FSWSP SDSP SDSP	RV32 RV64 RV128
11	>16*b*								

- 每个指令的编码空间，除了用于寄存器操作数的索引之外，还剩余众多位的编码空间，对于这些没有使用的编码空间，用户均可以加以利用。
- 另外对于某些特定的处理器实现，由于其往往不会实现所有的指令类型，对于没有实现的指令类型的编码空间，用户也可以加以利用。
- 有一些没有定义的指令类型组，用户也可以加以利用。

16.2.2 RISC-V 的预定义的 Custom 指令

为了便于用户对 RISC-V 进行扩展，RISC-V 架构甚至在 32 位的指令中预定义了 4 组 Custom 指令类型，每种 Custom 均有自己的 Opcode。如表 16-1 所示，custom-0、custom-1、custom-2 和 custom-3 共 4 种 Custom 指令类型。用户可以利用这 4 种指令类型扩展成为自定义的协处理器指令。蜂鸟 E200 处理器核便使用 Custom 指令扩展协处理器指令。

16.3 蜂鸟 E200 的协处理器接口 EAI

蜂鸟 E200 处理器核的协处理器机制借鉴了开源 RISC-V 处理器 Rocket Core 的协处理器接口 RoCC(Rocket Custom Coprocessor)，且接口信号定义非常类似.为了将其与原始的 RoCC 接口进行区分，命名为 EAI（Extension Accelerator Interface）。本节将以一个实际案例来详细阐述如何使用 EAI 接口和 Custom 指令扩展出蜂鸟 E200 协处理器。

注意：由于蜂鸟 E200 处理器核基于 Custom 指令进行协处理器扩展，因此本章将 Custom 指令也称为 EAI 指令。

16.3.1 EAI 指令的编码

32 位的 EAI 指令编码格式如图 16-1 所示。

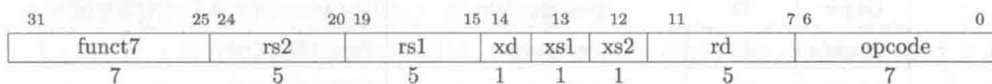


图 16-1　EAI 指令编码格式

（1）指令的第 0 位至第 6 位区间为 Opcode 编码段，根据表 16-1 中的编码规则使用编码 custom-0、custom-1、custom-2 和 custom-3 指令组。

（2）xs1、xs2 和 xd 比特位分别用于控制是否需要读源寄存器 rs1、rs2 和写目标寄存器 rd。

- 如果 xs1 位的值为 1，则表示该指令需要读取由 rs1 比特位索引的通用寄存器作为源

操作数1；如果xs1位的值为0，则表示该指令不需要源操作数1。

- 同理，如果xs2位的值为1，则表示该指令需要读取由rs2比特位索引的通用寄存器作为源操作数2；如果xs2位的值为0，则表示该指令不需要源操作数2。
- 如果xd位的值为1，则表示该指令需要写回结果至由rd比特位指示的目标寄存器；如果xd位的值为0，则表示该指令无须写回结果。

（3）指令的第25位～第31位为funct7区间，可作为额外的编码空间，用于编码更多的指令，因此一种Custom指令组可以使用funct7区间编码出128条指令，则4组Custom指令组共可以编码出512条两读一写（读取两个源寄存器，写回一个目标寄存器）的协处理器指令。如果有的协处理器指令仅读取一个源寄存器，或者无须写回目标寄存器，则可以使用这些无用的比特位（譬如rd比特位）来编码出更多的协处理器指令。

16.3.2 EAI接口信号

EAI接口信号如表16-3所示，EAI接口主要包含4个通道。

- 请求通道（Request Channel）：主要用于主处理器在EXU级将指令信息和源操作数派发给协处理器。
- 反馈通道（Response Channel）：主要用于协处理器反馈主处理器告知其已经完成了该指令，并将结果写回主处理器。
- 存储器请求通道（Memory Request Channel）：主要用于协处理器向主处理器发起存储器读写请求。
- 存储器反馈通道（Memory Response Channel）：主要用于主处理器向协处理器返回存储器读写结果。

表16-3　EAI接口信号

通　道	方向	宽度	信 号 名	介　绍
请求通道	Output	1	eai_req_valid	主处理器向协处理器发送指令请求信号
	Input	1	eai_req_ready	协处理器向协处理器返回指令接收信号
	Output	32	eai_req_instr	该Custom指令的32比特完整编码
	Output	32	eai_req_rs1	源操作数1的值
	Output	32	eai_req_rs2	源操作数2的值
	Output	2	eai_req_itag	该指令的派发标号
反馈通道	Input	1	eai_rsp_valid	协处理器向主处理器发送反馈请求信号
	Output	1	eai_rsp_ready	主处理器向协处理器返回反馈接收信号
	Input	32	eai_rsp_wdat	返回计算结果的值
	Input	2	eai_rsp_itag	返回该指令的派发标号
	Input	1	eai_rsp_err	返回该指令的错误标志

续表

通道	方向	宽度	信号名	介绍
存储器请求通道	Input	1	eai_icb_cmd_valid	协处理器向主处理器发送存储器读写请求信号
	Output	1	eai_icb_cmd_ready	主处理器向协处理器返回存储器读写接收信号
	Input	32	eai_icb_cmd_addr	存储器读写地址
	Input	1	eai_icb_cmd_read	存储器读或是写的指示
	Input	32	eai_icb_cmd_wdata	写存储器的数据
	Input	4	eai_icb_cmd_wmask	写存储器的字节掩码
存储器反馈通道	Output	1	eai_icb_rsp_valid	主处理器向协处理器发送存储器读写反馈请求信号
	Input	1	eai_icb_rsp_ready	协处理器向协处理器返回存储器读写反馈接收信号
	Output	32	eai_icb_rsp_rdata	存储器读反馈的数据
	Output	1	eai_icb_rsp_err	存储器读写反馈的错误标志
存储器占用	Input	1	eai_mem_holdup	协处理器需要独占存储器访问通道的指示信号

16.3.3 EAI 流水线接口

基于 EAI 接口的协处理器在蜂鸟 E200 流水线中位置如图 16-2 所示。

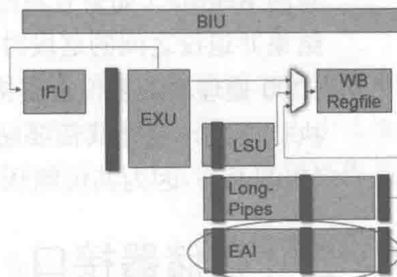


图 16-2 EAI 接口协处理器在流水线中的位置

整个 EAI 指令的执行过程如下。

- 主处理器的译码（Decode）单元在 EXU 级对指令的 Opcode 进行译码，判断其是否属于任意一种 Custom 指令组。
- 如果主处理器译码判断该指令属于 Custom 指令，则继续依据指令编码的 xs1 和 xs2 位判断是否需要读取源寄存器。如需要读取，则在 EXU 级读取通用寄存器组（Regfile）读出源操作数。
- 主处理器会维护数据依赖的正确性，譬如该指令需要读取的源寄存器与之前正在执行的某条指令存在着先写后读(RAW)的依赖性，则处理器流水线会暂停直至该 RAW 解除。另外，主处理器会依据指令编码的 xd 位判断该 Custom 指令是否需要写回结果至通用寄存器组，如需要写回，则会将目标寄存器的索引信息存储在主处理器的流水线控制模块中，直至写回完成，以便供后续的指令进行数据依赖性的判断。
- 经过上述的步骤之后，主处理器在 EXU 级通过 EAI 接口的请求通道派发给外部的协处理器，派发的信息包括指令的编码信息、两个源操作数的值（由于蜂鸟 E200

是 32 位架构，则两个源操作数均为 32 位宽）和该指令的派发标号（Dispatch ITag）。在 10.2.3 节中曾经介绍过蜂鸟 E200 处理器属于“乱序执行-顺序写回”。因此该派发标号主要用于追踪指令的派发顺序，协处理器收到 ITag 后需要携带该 ITag，直至写回结果并连同 ITag 返还给主处理器。

- 协处理器通过请求通道接受指令之后，需对指令做进一步的译码，并进行实际的执行操作。协处理器可以单拍便返回结果，也可以多拍之后再返回结果。对于多拍返回结果的指令，其行为可以是阻塞式的，也可以是流水线式。由于协处理器和主处理器的请求通道采取的是 Valid-Ready 方式的同步握手接口，只要协处理器能够接受指令，就可以将 Ready 信号拉高。因此只要协处理器能够先后连续接受多条指令，则主处理器就可以先后连续发送多条指令至协处理器。
- 协处理器在完成执行后，通过 EAI 接口的反馈通道将结果反馈给主处理器。如果协处理器接受并执行了多条指令，则需要保证其在 Response Channel 反馈结果的顺序必须与其在请求通道接受指令时的顺序一致，即按序写回，因此反馈通道需要包括该指令的 ITag 并遵循其顺序。如果是需要写回结果的指令，则反馈通道还需包含返回结果的值。
- 主处理器在收到反馈通道的反馈结果之后，则将此次指令从流水线中退役并将结果写回 Regfile（如果有写回需求）。Custom 指令从被发送至协处理器到协处理器反馈结果并退役之间的这段时间，我们称之为滞外指令（Outstanding Instructions）。蜂鸟 E200 处理器最多仅能支持不超过 4 条以上的滞外指令。因此如果协处理器是流水线执行方式，但是其需要超过 4 个周期以上才能反馈结果，那么流水线会出现空泡（Bubble），因为其仅能从主处理器收到 4 条指令。

16.3.4 EAI 存储器接口

支持协处理器访问存储器资源可以扩大协处理器的类型范围，使得协处理器不仅限于运算指令类型。在处理器的 LSU 模块中为 EAI 协处理器预留了专用的访问接口，如图 16-3 所示。因此基于 EAI 接口的协处理器可以访问主处理器能够寻址的数据存储器资源，包括 ITCM、DTCM、系统存储总线、系统设备总线以及快速 IO 接口等。

EAI 指令访问存储器资源的实现机制如下。

- 主处理器的 LSU 为 EAI 协处理器预留的专用访问通道基于 ICB 总线标准。
- 为了防止后续指令访问存储器和 EAI 协处理器访问存储器形成竞争死锁。协处理器在接收到 EAI 的请求通道发送过来的指令后进行译码，如果发现是需要访问存储器资源的协处理器指令，则需立即将存储器独占信号拉高（eai_mem_holdup），主处理器将会阻止后续的指令继续访问存储器资源。

- 协处理器需要访问存储器是通过 ICB 的存储器请求通道向主处理器的 LSU 发起请求。存储器请求通道中的信息包括需要访问的存储器地址，访问是读或是写操作。如果是读操作，意味着对主处理器进行 32 位对齐的一次读操作；如果是写操作，则通过存储器请求通道中的写数据（wdata）和字节掩码（wmask）控制写操作的数据和粒度。

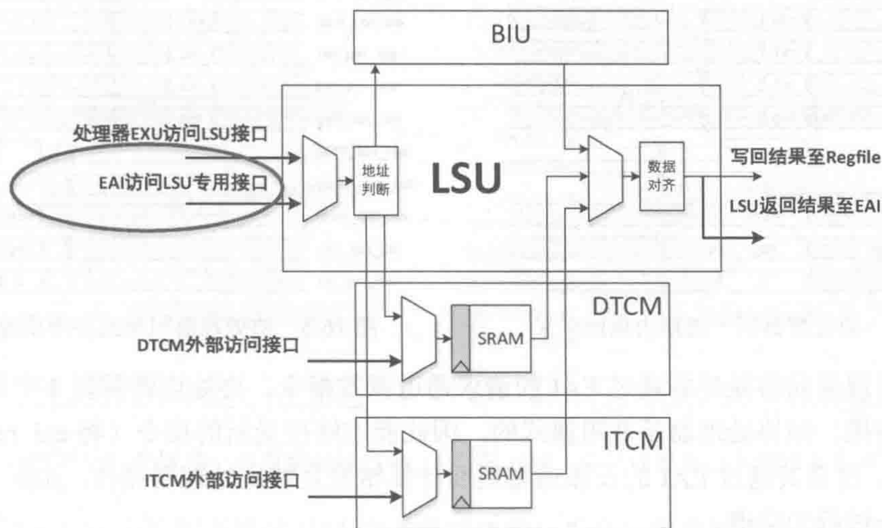


图 16-3　LSU 为 EAI 预留专用访问通道

- 主处理器的 LSU 在完成存储器读写操作后，通过 ICB 的存储器反馈通道向协处理器反馈。如果是读操作，存储器反馈通道中的信息包括返回的读数据，和本次读操作是否发生了错误；如果是写操作，存储器反馈通道中的信息仅包含本次写操作是否发生了错误。
- 由于协处理器和主处理器 LSU 接口的 ICB 总线采取的是 Valid-Ready 方式的同步握手接口，因此只要主处理器的 LSU 能够先后连续多次接受存储器访问操作，则协处理器可以先后连续多次发送多个存储器读写请求。
- 协处理器在完成对存储器的访存之后，需将存储器独占信号拉低（eai_mem_holdup），主处理器将会释放 LSU 允许后续的指令继续访问存储器资源。

16.3.5 EAI 接口时序

本节将描述 EAI 接口的若干典型时序，以帮助读者理解 EAI 接口。

- 主处理器向协处理器通过 EAI 的请求通道派发指令，协处理器在同一个周期即返回结果，但是主处理器不能接收，下一周期才能接收此结果，如图 16-4 所示。
- 主处理器向协处理器通过 EAI 的请求通道派发指令，协处理器需要多个周期才能返

回结果，且协处理器是阻塞式的，因此其不能接受新的指令（将 eai_req_ready 拉低），直至其通过 EAI 的反馈通道返回计算结果且被主处理器接收，如图 16-5 所示。

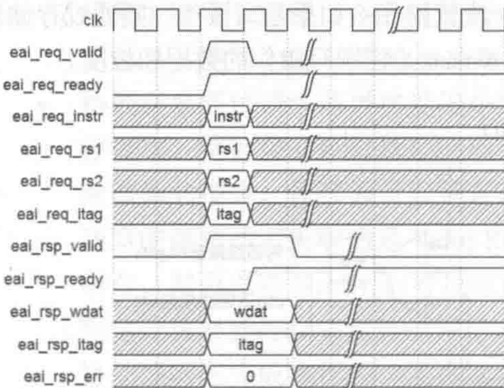


图 16-4 协处理器同一周期内返回结果

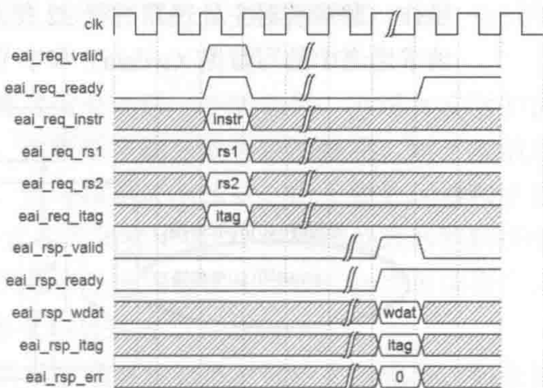


图 16-5 协处理器阻塞式多周期返回结果

- 主处理器向协处理器通过 EAI 的请求通道派发指令，协处理器需要 4 个周期才能返回结果。但协处理器是非阻塞式的，因此能连续接受新的指令（将 eai_req_ready 拉高），直至其通过 EAI 的反馈通道返回计算结果且被主处理器接收，如图 16-6 所示，整个过程无空泡。
- 主处理器向协处理器通过 EAI 的请求通道派发指令，协处理器需要 8 个周期才能返回结果，但协处理器是非阻塞式的，因此能连续接受新的指令（将 eai_req_ready 拉高）。但是主处理器最多只能发送 4 条滞外指令，在 EAI 的反馈通道返回计算结果且被主处理器接收之前会出现 4 个空泡周期，如图 16-7 所示。

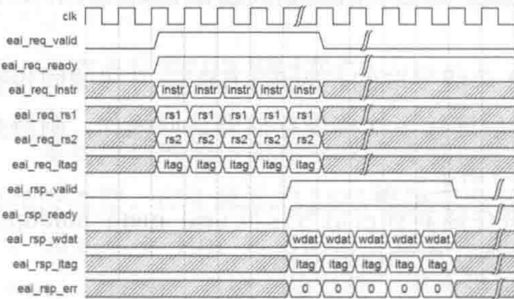


图 16-6 协处理器非阻塞式
连续接收 4 条指令无空泡周期

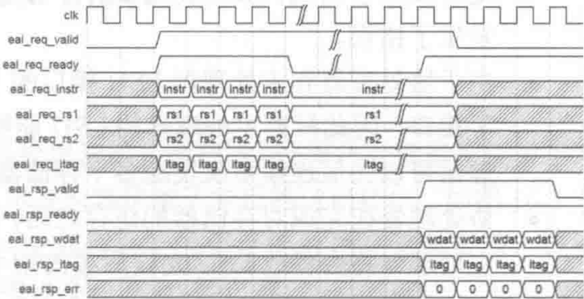


图 16-7 协处理器非阻塞式
连续接收 4 条指令有空泡周期

- 主处理器向协处理器通过 EAI 的请求通道派发指令，协处理器译码出该指令需要访问存储器，则将存储器独占信号拉高（eai_mem_holdup），协处理器通过存储器请求通道向主处理器 LSU 发起读写请求，主处理器通过存储器反馈通道在下一个周期返回结果，协处理器继而将存储器独占信号拉低，如图 16-8 所示。

- 主处理器向协处理器通过 EAI 的请求通道派发指令，协处理器译码出该指令需要访问存储器，则将存储器独占信号拉高（eai_mem_holdup），协处理器通过存储器请求通道向主处理器 LSU 发起读写请求，主处理器通过存储器反馈通道在多个周期之后返回结果，协处理器继而将存储器独占信号拉低，如图 16-9 所示。

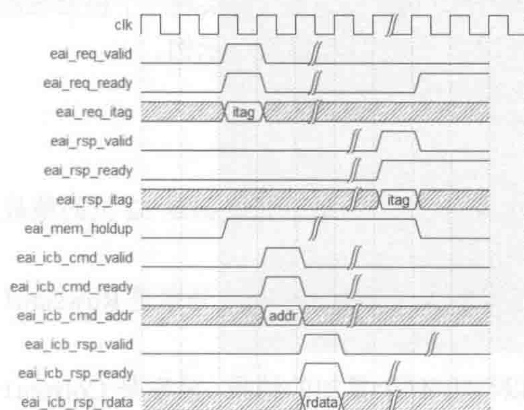


图 16-8 协处理器访问存储器一个周期返回结果

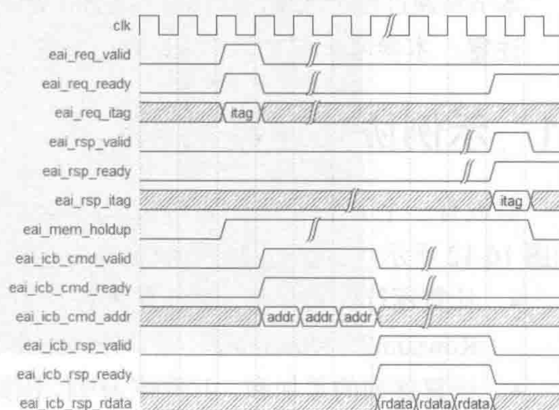


图 16-9 协处理器访问存储器多个周期返回结果

- 主处理器向协处理器通过 EAI 的请求通道派发指令，协处理器译码出该指令需要访问存储器，则将存储器独占信号拉高（eai_mem_holdup），协处理器通过存储器请求通道向主处理器 LSU 发起读写请求，主处理器通过存储器反馈通道在多个周期之后返回结果，但是结果指示读写存储器是发生了错误，协处理器继而将存储器独占信号拉低，如图 16-10 所示。
- 主处理器向协处理器通过 EAI 的请求通道派发指令，协处理器译码出该指令是个非法指令，其通过反馈通道返回错误标志，如图 16-11 所示。

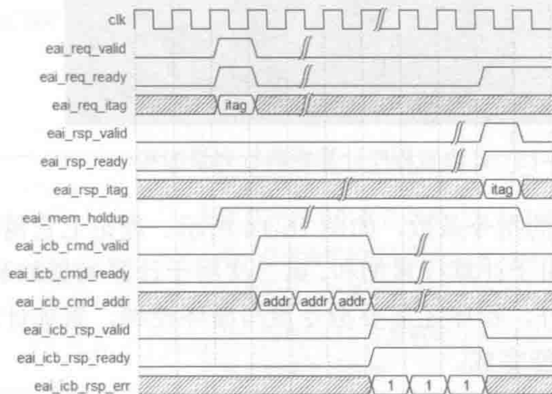


图 16-10 协处理器访问存储器多个周期返回错误标志

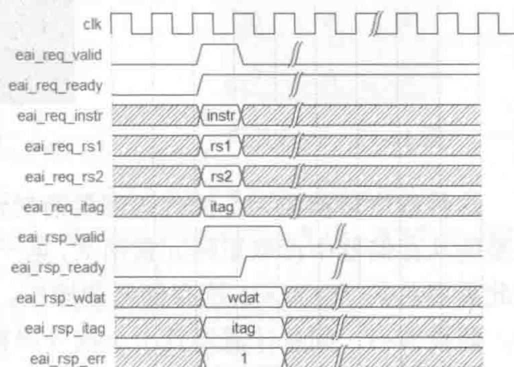


图 16-11 协处理器非法指令返回错误标志

16.4 蜂鸟 E200 的协处理器参考示例

本节将通过参考示例，阐述蜂鸟 E200 处理器如何使用 EAI 接口定义并实现一个协处理器。

注意：本参考示例来源于 Rocket-Chip 开源项目的一个协处理器教学示例。

16.4.1 示例协处理器需求

假设有一个 *N* 行 *N* 列的矩阵按顺序存储在存储器中，矩阵的每个元素都是 32 位的整数，如图 16-12 所示（3 行 3 列的示例矩阵）。需要对该矩阵进行如下操作。

- 计算逐行的累加和，由于有 *N* 行，因此可以得出 *N* 个累加的结果，分别是 Rowsum1，Rowsum2，Rowsum3，…，Rowsum<*N*>。
- 计算逐列的累加和，由于有 *M* 列，因此可以得出 *M* 个累加的结果，分别是 Colsum1，Colsum2，Colsum3，…，Colsum<*M*>。

如果采用常规的程序进行计算，则需要采用循环方式，按行读取各个元素，然后将各个元素相加得到各行的累加和，然后再采取循环方式，按列读取各个元素，然后将各个元素相加得到各列的累加和，如图 16-13 所示。

1	2	3
4	5	6
7	8	9

图 16-12　3×3 矩阵

```
Start_Timer();

for (i = 0; i < msize; i++){
  rawsum[i] = 0;
  for (j = 0; j < msize; j++){
    rawsum[i] = rawsum[i] + matrix[i][j];
  }
}

for (i = 0; i < msize; i++){
  colsum[i] = 0;
  for (j = 0; j < msize; j++){
    colsum[i] = colsum[i] + matrix[j][i];
  }
}

Stop_Timer();

User_Time = End_Time - Begin_Time;
```

图 16-13　C 语言程序计算矩阵行列累加和

这种程序转换成为汇编代码需要消耗较多的指令条数，如图 16-14 所示。理论上它需要完整的从存储器中读取矩阵元素两次，第一次用于计算行累加和，第二次用于计算列累加和，因此需要总共 $N\times N\times 2$ 次存储器读操作。此外，程序还需要指令参与循环控制，累加计算等。假设 N=3，则总计需要几十个指令周期才能完成。

```
80002150:   429c          lw  a5,0(a3)
80002152:   42c8          lw  a0,4(a3)
80002154:   468c          lw  a1,8(a3)
80002156:   06b1          addi    a3,a3,12
80002158:   97aa          add a5,a5,a0
8000215a:   97ae          add a5,a5,a1
8000215c:   c21c          sw  a5,0(a2)
8000215e:   0611          addi    a2,a2,4
80002160:   fed818e3    bne a6,a3,80002150 <main+0x68>
80002164:   474c          lw  a1,12(a4)
80002166:   4f10          lw  a2,24(a4)
80002168:   4785          li  a5,1
8000216a:   0040          addi    s0,sp,4
8000216c:   97ae          add a5,a5,a1
8000216e:   86a2          mv  a3,s0
80002170:   97b2          add a5,a5,a2
80002172:   c29c          sw  a5,0(a3)
80002174:   1028          addi    a0,sp,40
80002176:   0711          addi    a4,a4,4
80002178:   0691          addi    a3,a3,4
8000217a:   00e50c63    beq a0,a4,80002192 <main+0xaa>
8000217e:   431c          lw  a5,0(a4)
80002180:   474c          lw  a1,12(a4)
80002182:   4f10          lw  a2,24(a4)
80002184:   0711          addi    a4,a4,4
80002186:   97ae          add a5,a5,a1
80002188:   97b2          add a5,a5,a2
8000218a:   c29c          sw  a5,0(a3)
8000218c:   0691          addi    a3,a3,4
8000218e:   fee518e3    bne a0,a4,8000217e <main+0x96>
```

图 16-14　C 语言编译所得汇编程序

16.4.2　示例协处理器指令

为了提高性能和能效比，可以将此矩阵操作定义成为协处理器指令进行加速。如表 16-4 所示，定义了 3 条指令，分别是 SETUP、ROWSUM 和 COLSUM。

表 16-4　示例协处理器指令

协处理器指令	介　绍	编　码
SETUP	SETUP 指令定义矩阵的大小，并指明存储器中连续存储矩阵元素地址区间的第一个地址	• Opcode 指明使用 Custom0 指令组 • xd 位的值为 0，表示此指令并不需要写回任何结果 • xs1 位的值为 1，表示此指令需要读取操作数 rs1。操作数 rs1 的值为连续存储矩阵元素地址区间的第一个地址 • xs2 位的值为 1，表示此指令需要读取操作数 rs2。操作数 rs2 的值为指示矩阵的大小的 N 值 • funct7 的值为 0，该值编码 SETUP 指令
ROWSUM	ROWSUM 指令指示协处理器计算指定行的累加值，并通过结果寄存器返回累加值，如果发生了溢出，则返回 32 位整数能表示的最大饱和值	• Opcode 指明使用 Custom0 指令组 • xd 位的值为 1，表示此指令需要通过写回结果至 rd 寄存器 • xs1 位的值为 1，表示此指令需要读取操作数 rs1。操作数 rs1 的值为指定行的行号（从 1 开始编号） • xs2 位的值为 0，表示此指令不需要读取操作数 rs2 • funct7 的值为 1，该值编码 ROWSUM 指令

续表

协处理器指令	介　绍	编　码
COLSUM	COLSUM 指令指示协处理器计算指定列的累加值，并通过结果寄存器返回累加值，如果发生了溢出，则返回 32 位整数能表示的最大饱和值	• Opcode 指明使用 Custom0 指令组 • xd 位的值为 1，表示此指令需要通过写回结果至 rd 寄存器 • xs1 位的值为 1，表示此指令需要读取操作数 rs1。操作数 rs1 的值为指定列的列号（从 1 开始编号） • xs2 位的值为 0，表示此指令不需要读取操作数 rs2 • funct7 的值为 2，该值编码 COLSUM 指令
注意：以上的几条指令需要按照特定的顺序编写 • 首先使用 SETUP 指令指定矩阵的大小及相关参数 • 然后使用多条 ROWSUM 指令分别计算各行的累加和 • 最后使用多条 COLSUM 指令分别计算各列的累加和		

16.4.3 示例协处理器实现

示例协处理器的硬件实现框图如图 16-15 所示，其主要由控制模块和累加模块组成。控制模块主要负责和主处理器的 EAI 接口交互，并调用累加器进行累加运算。

累加器框图如图 16-16 所示，其主要负责累加运算。

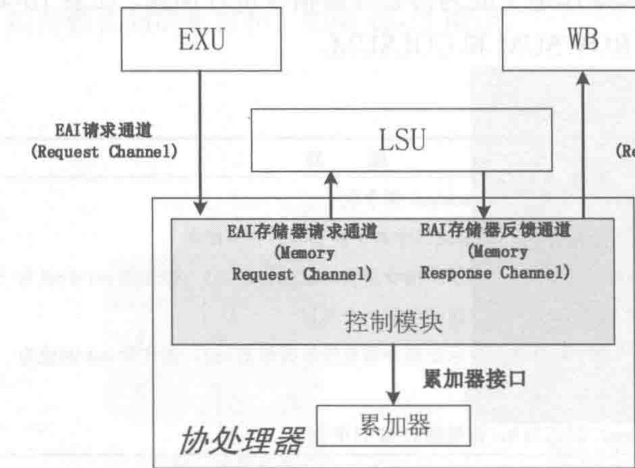


图 16-15　示例协处理器控制模块框图

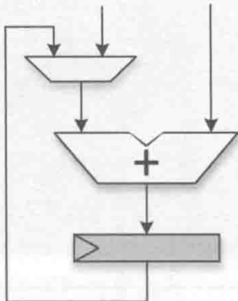


图 16-16　示例协处理器累加器模块框图

示例协处理器实现方案的处理过程如表 16-5 所示。

表 16-5　示例协处理器实现方案

步　骤	指 令 行 为	硬 件 行 为
步骤 1	使用 SETUP 指令指定需要运算的矩阵参数	• 控制模块使用参数设置合适的行为

续表

步骤	指令行为	硬件行为
步骤 2	使用多条 ROWSUM 指令分别计算各行的累加和	• 为了能够复用读取的数据，在硬件模块中分配一个行大小的寄存器缓冲（Buffer），称之为 buf_row，用于存储上一次处理的行的元素值 • ROWSUM 指令将触发协处理器从存储器地址中逐个读取该行的元素值，且每次读取回来的数值被送给累加器进行累加，完全读取 *N* 个元素后的累加结果即为该行的累加和 • 从第二行开始，每次读取回来的数值不仅需要使用累加器计算该行的累加值，还需使用新读取的数据和上一行的数据（存储在 buf_row 之中）累加，并且将结果继续暂存在 buf_row 之中 • 经过 *N* 个 ROWSUM 指令之后，不仅每行的累加和已经被计算出，而且每一列的累加和也被计算出存储在 buf_row 之中
步骤 3	使用多条 COLSUM 指令分别计算各列的累加和	• 由于在步骤 2 中已经计算出了各列的累加和，存储在 buf_row 之中。因此本步骤无需任何的存储器访问和计算操作，直接从 buf_row 中取出所需的计算结果即可

以图 16-12 中的矩阵实例为例，使用协处理器后整个计算过程的数值如表 16-6 所示。

表 16-6 示例协处理器计算过程的数值

累加和	结果	存储器访问次数	列 1 部分和	列 2 部分和	列 3 部分和
ROWSUM 指令计算行 1 累加和	6	3	1	2	3
ROWSUM 指令计算行 2 累加和	15	3	5	7	9
ROWSUM 指令计算行 3 累加和	24	3	12	15	18
COLSUM 指令计算列 1 累加和	12	0	12	15	18
COLSUM 指令计算列 2 累加和	15	0	12	15	18
COLSUM 指令计算列 3 累加和	18	0	12	15	18

16.4.4 示例协处理器性能

以图 16-12 中的矩阵实例为例，理论分析示例协处理器的性能如下。

- SETUP 指令可以在一个周期内完成。
- 假设矩阵存储在 DTCM 区间内，可以通过一个周期从 DTCM 中读回一个矩阵元素，则计算第一行的累加和需要 3 个时钟周期访问 3 个存储器地址，采用流水线的方式连续读取和运算（读取一个元素后的累加计算和下一次读取操作重叠），则总共需要 4 个周期完成第一行的 ROWSUM 指令。
- 从第二行开始，不仅需要计算该行的累加和，还需要累加计算得到 row_buf 中每一列的部分累加和，因此需要额外的 3 个周期，则总共需要 7 个周期完成第二行和第

三行的 ROWSUM 指令。

- 每一个 COLSUM 指令可以直接从 row_buf 中取出计算结果，可以在一个周期内完成。
- 综上，完成全部操作需要 7 条指令，1+4+7+7+1+1+1=22 个时钟周期完成。

与普通的 C 语言程序实现相比的性能对比如表 16-7 所示，在性能和能效比方面就能带来 3～5 倍的提升。

表 16-7　　使用示例协处理器与不使用之性能对比

	普通的 C 语言程序	使用示例协处理器
动态执行指令数	66	7
动态执行周期数	80	22
读取 ITCM 取指令次数	66	7
读取 DTCM 取数据次数	18	9

16.4.5　示例协处理器代码

示例协处理器的代码包括两部分内容。

- 示例协处理器的 Verilog RTL 实现代码。
- 使用协处理器指令的软件代码。

本书在此不予详述其代码，有兴趣的读者可以自行尝试实现或者联系作者。

第三部分

使用 Verilog 进行仿真和在 FPGA SoC 原型上运行软件

第 17 章　冒个烟先
——运行 Verilog 仿真测试



本章将介绍 E200 开源平台如何运行 Verilog 仿真测试。注意：为了能够跟随本章介绍的内容重现相关仿真环境，需要具备 Linux 命令行以及 Makefile 脚本的基本知识。

17.1 E200 开源项目的代码层次结构

蜂鸟 E200 的开源平台托管于著名开源网站 GitHub。GitHub 是一个著名的免费项目的托管网站，任何用户无须注册即可从网站上下载源代码，很多的开源项目均将源代码托管于此，E200 的项目网址请在 GitHub 中搜索“e200_opensource”，如图 17-1 所示。

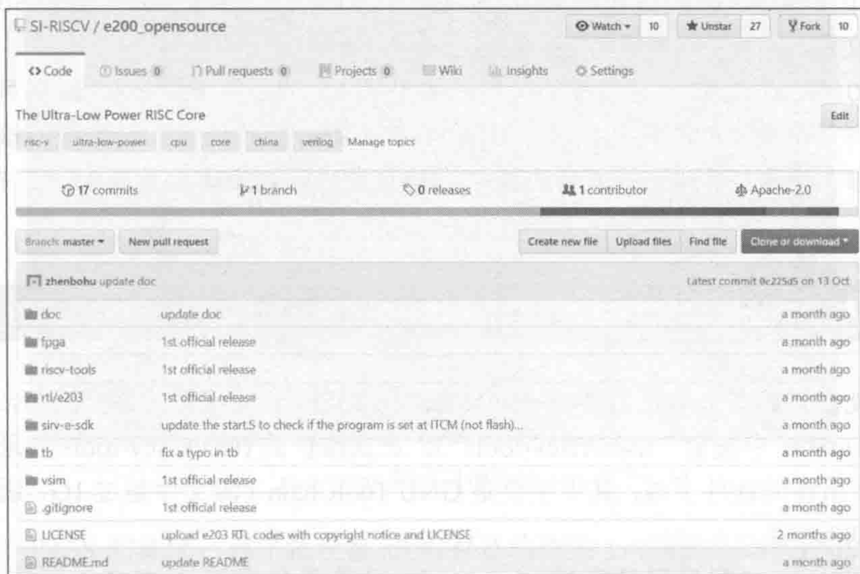


图 17-1 蜂鸟 E200 开源 GitHub 网址

在该网址的 e200_opensource 目录下，文件的层次结构如下所示。

```
e200_opensource
     |----rtl                          // 存放 RTL 的目录
          |----e203                    // E203 核和 SoC 的 RTL 目录
               |----general            // 存放一些公用的通用 RTL 代码
               |----core               // 存放 e203 Core 的 RTL 代码
               |----fab                // 存放总线 bus fabric 的 RTL 代码
               |----subsys             // 存放完整子系统顶层的 RTL 代码
               |----mems               // 存放 memory 模块的 RTL 代码
               |----perips             // 存放外设 peripherals 模块的 RTL 代码
               |----debug              // 存放 debug 相关模块的 RTL 代码
               |----fpga               // 存放 FPGA 实现的 RTL 代码
      |----tb                          // 存放 Verilog TestBench （测试平台）的目录
```

```
        |----tb_top.v          // 简单的 Verilog TestBench 顶层文件
    |----vsim                  // 运行仿真的目录
        |----bin               // 存放脚本的文件夹子
        |----Makefile          // 运行的 Makefile
        |----run               // 运行目录
    |----fpga                  // 存放 FPGA 项目和脚本的目录
    |----riscv-tools           // 存放所需 riscv-tools 的目录
        |----riscv-fesvr       // 用于编译指令模拟器 Spike 的源代码
        |----riscv-isa-sim     // 用于编译指令模拟器 Spike 的源代码
        |----riscv-tests       // 存放一些测试用例的目录
    |----doc                   // 保存文档的目录
    |----README.md             // 说明文件
```

rtl 目录下包含了大量的源代码，主要为目前开源 Core（譬如 E203）和配套 SoC 的 Verilog RTL 源代码文件。

有关 E201/E203/E205 处理器核的区别请参见第 4.3 节中关于蜂鸟 E200 系列的简介，每个处理器核有一个专属的文件夹目录存放期 Verilog RTL 源代码。关于处理器核部分的具体代码列表请参见第 5.5 节详述，配套的 SoC 的信息和代码列表请参见第 18.2.2 节。

17.2 E200 开源项目的测试用例

读者可能注意到在上节中所述的 riscv-tools 与 RISC-V 架构正式维护的 riscv-tools 项目同名（请在 GitHub 中搜索"riscv/riscv-tools"）。正式维护的 riscv/riscv-tools 目录下包括了所有的 RISC-V 所需的软件工具，其中主要是 GNU ToolChain（源文件超过 1G，因此下载需要相当长的时间）。

e200_opensource 项目目录下的 riscv-tools 目录仅包含编译 Spike 所需的源代码和 riscv-tests，放置该目录于此是因为正式维护的 riscv/riscv-tools 在不断更新，而 e200_opensource 下的 riscv-tools 仅用于支持运行自测试用例，因此无须使用最新版本。此外，还进行了适当修改（譬如，在 riscv-tests 里面添加了更多的测试用例，生成更多的 log 文件），同时去除了 GNU ToolChain，使得文件夹很小，方便读者快速下载使用。

17.2.1 riscv-tests 自测试用例

所谓自测试用例（Self-Check Testcase），是一种能够自我检测运行成功还是失败的测试程序。riscv-test 是由 RISC-V 架构开发者维护的开源项目，包含一些测试处理器是否符合指令集架构定义的测试程序，这些测试程序均由汇编语言编写。

注意：e200_opensource 下的 riscv-tests 目录复制于原始的 riscv-test 项目（请在 GitHub

中搜索“riscv/riscv-tests”），在此基础上添加了更多的测试用例和生成更多的 log 文件。

此类汇编测试程序里用某些宏定义组织成程序点，测试指令集架构中定义的指令，如图 17-2 所示。测试 add 指令（源代码文件为 isa/rv64ui/add.S），通过让 add 指令执行两个数据的相加（譬如 0x0000003 和 0x00000007），设定它期望的结果（譬如 0x0000000a）。然后使用比较指令加以判断，假设 add 指令的执行结果的确与期望的结果相等，则程序继续执行；假设与期望的结果不相等，则程序直接使用 jump 指令跳到 TEST_FAIL 地址；假设所有的测试点都通过，则程序一直执行到 TEST_PASS 地址。

```
RVTEST_CODE_BEGIN

  TEST_RR_OP( 2,  add, 0x00000000, 0x00000000, 0x00000000 );
  TEST_RR_OP( 3,  add, 0x00000002, 0x00000001, 0x00000001 );
  TEST_RR_OP( 4,  add, 0x0000000a, 0x00000003, 0x00000007 );

  TEST_RR_OP( 5,  add, 0xffffffffffff8000, 0x0000000000000000, 0xffffffffffff8000 );
  TEST_RR_OP( 6,  add, 0xffffffff80000000, 0xffffffff80000000, 0x00000000 );
  TEST_RR_OP( 7,  add, 0xffffffff7fff8000, 0xffffffff80000000, 0xffffffffffff8000 );

  TEST_RR_OP( 8,  add, 0x0000000000007fff, 0x0000000000000000, 0x0000000000007fff );
  TEST_RR_OP( 9,  add, 0x000000007fffffff, 0x000000007fffffff, 0x0000000000000000 );
  TEST_RR_OP( 10, add, 0x0000000080007ffe, 0x000000007fffffff, 0x0000000000007fff );

  TEST_RR_OP( 11, add, 0xffffffff80007fff, 0xffffffff80000000, 0x0000000000007fff );
  TEST_RR_OP( 12, add, 0x000000007fff7fff, 0x000000007fffffff, 0xffffffffffff8000 );

  TEST_RR_OP( 13, add, 0xffffffffffffffff, 0x0000000000000000, 0xffffffffffffffff );
  TEST_RR_OP( 14, add, 0x0000000000000000, 0xffffffffffffffff, 0x0000000000000001 );
  TEST_RR_OP( 15, add, 0xfffffffffffffffe, 0xffffffffffffffff, 0xffffffffffffffff );

  TEST_RR_OP( 16, add, 0x0000000080000000, 0x0000000000000001, 0x000000007fffffff );

  TEST_RR_SRC1_EQ_DEST( 17, add, 24, 13, 11 );
  TEST_RR_SRC2_EQ_DEST( 18, add, 25, 14, 11 );
  TEST_RR_SRC12_EQ_DEST( 19, add, 26, 13 );
```

图 17-2　riscv-tests 测试用例测试 add 指令片段

在 TEST_PASS 的地址，程序将设置 x3 寄存器的值为 1，而在 TEST_FAIL 的地址，程序将 x3 寄存器的值设置为非 1 值。因此，最终可以通过判断 x3 的值来界定程序的运行结果到底是成功还是失败。

17.2.2　编译 ISA 自测试用例

riscv-tests 中的指令集架构（ISA）测试用例都是使用汇编语言编写，为了在仿真阶段能够被处理器执行，需要将这些汇编程序编译成二进制代码。在 e200_opensource 的以下目录（generated 文件夹）下，已经预先上传了一组编译完毕的可执行文件和反汇编文件，以及能够被 Verilog 的 readmemh 函数读入的文件。

```
e200_opensource
    |----riscv-tools                // 存放所需 riscv-tools 的目录
        |----riscv-tests            // 存放一些测试用例的目录
            |----isa
```

```
                |----generated                      // 编译好的 tests 文件夹
                    |----rv32ui-p-addi              // 编译出的 elf 文件
                    |----rv32ui-p-addi.dump         // 反汇编文件
                    |----rv32ui-p-addi.verilog      // 可被 Verilog 的 readmemh
                                                    // 函数读入的文件
                ......
```

譬如，反汇编文件（譬如 rv32ui-p-addi.dump）的内容如图 17-3 所示。

譬如，Verilog 的 readmemh 函数能够读入的文件（譬如 rv32ui-p-addi.verilog）内容如图 17-4 所示。

```
rv32ui-p-add:     file format elf32-littleriscv


Disassembly of section .text.init:

80000000 <_start>:
80000000:   a081                    j   80000040 <reset_vector>
80000002:   0001                    nop

80000004 <trap_vector>:
80000004:   34202f73            csrr    t5,mcause
80000008:   4fa1                    li  t6,8
8000000a:   03ff0663            beq t5,t6,80000036 <write_tohost>
8000000e:   4fa5                    li  t6,9
80000010:   03ff0363            beq t5,t6,80000036 <write_tohost>
80000014:   4fad                    li  t6,11
80000016:   03ff0063            beq t5,t6,80000036 <write_tohost>
8000001a:   80000f17            auipc   t5,0x80000
8000001e:   fe6f0f13            addi    t5,t5,-26 # 0 <_start-0x80000000>
80000022:   000f0363            beqz    t5,80000028 <trap_vector+0x24>
80000026:   8f02                    jr  t5
80000028:   34202f73            csrr    t5,mcause
8000002c:   000f5363            bgez    t5,80000032 <handle_exception>
80000030:   a009                    j   80000032 <handle_exception>

80000032 <handle_exception>:
80000032:   5391e193            ori gp,gp,1337

80000036 <write_tohost>:
80000036:   00001f17            auipc   t5,0x1
8000003a:   fc3f2523            sw  gp,-54(t5) # 80001000 <tohost>
8000003e:   bfe5                    j   80000036 <write_tohost>

80000040 <reset_vector>:
80000040:   f1402573            csrr    a0,mhartid
80000044:   e101                    bnez    a0,80000044 <reset_vector+0x4>
80000046:   4181                    li  gp,0
80000048:   00000297            auipc   t0,0x0
8000004c:   fbc28293            addi    t0,t0,-68 # 80000004 <trap_vector>
80000050:   30529073            csrw    mtvec,t0
80000054:   80000297            auipc   t0,0x80000
80000058:   fac28293            addi    t0,t0,-84 # 0 <_start-0x80000000>
8000005c:   00028e63            beqz    t0,80000078 <reset_vector+0x38>
80000060:   10529073            csrw    stvec,t0
```

图 17-3 反汇编文件内容片段


```
@00000000
81 A0 01 00 73 2F 20 34 A1 4F 63 06 FF 03 A5 4F
63 03 FF 03 AD 4F 63 00 FF 03 17 0F 00 80 13 0F
6F FE 63 03 0F 00 02 8F 73 2F 20 34 63 53 0F 00
09 A0 93 E1 91 53 17 1F 00 00 23 25 3F FC E5 BF
73 25 40 F1 01 E1 81 41 97 02 00 00 93 82 C2 FB
73 90 52 30 97 02 00 80 93 82 C2 FA 63 8E 02 00
73 90 52 10 B7 B2 00 00 93 82 92 10 73 90 22 30
73 23 20 30 E3 9F 62 FA 73 50 00 30 97 02 00 00
93 82 42 01 73 90 12 34 73 25 40 F1 73 00 20 30
81 40 01 41 33 8F 20 00 81 4E 89 41 63 1D DF 37
85 40 05 41 33 8F 20 00 89 4E 8D 41 63 15 DF 37
8D 40 1D 41 33 8F 20 00 A9 4E 91 41 63 1D DF 35
81 40 37 81 FF FF 33 8F 20 00 B7 8E FF FF 95 41
63 13 DF 35 B7 00 00 80 01 41 33 8F 20 00 B7 0E
00 80 99 41 63 19 DF 33 B7 00 00 80 37 81 FF FF
33 8F 20 00 B7 8E FF 7F 9D 41 63 1E DF 31 81 40
37 81 00 00 13 01 F1 FF 33 8F 20 00 B7 8E 00 00
93 8E FE FF A1 41 63 10 DF 31 B7 00 00 80 93 80
F0 FF 01 41 33 8F 20 00 B7 0E 00 80 93 8E FE FF
A5 41 63 12 DF 2F B7 00 00 80 93 80 F0 FF 37 81
00 00 13 01 F1 FF 33 8F 20 00 B7 8E 00 80 93 8E
EE FF A9 41 63 11 DF 2D B7 00 00 80 37 81 00 00
13 01 F1 FF 33 8F 20 00 B7 8E 00 80 93 8E FE FF
AD 41 63 12 DF 2B B7 00 00 80 93 80 F0 FF 37 81
FF FF 33 8F 20 00 B7 8E FF 7F 93 8E FE FF B1 41
63 13 DF 29 81 40 13 01 F0 FF 33 8F 20 00 93 0E
F0 FF B5 41 63 19 DF 27 93 00 F0 FF 05 41 33 8F
20 00 81 4E B9 41 63 1D DF 27 93 00 F0 FF 13 01
F0 FF 33 8F 20 00 93 0F F0 FF BD 41 63 15 DF 25
85 40 37 01 00 80 13 01 F1 FF 33 8F 20 00 B7 0E
00 80 C1 41 63 19 DF 23 B5 40 2D 41 8A 90 E1 4E
C5 41 63 92 D0 23 B9 40 2D 41 06 91 E5 4E C9 41
63 1B D1 21 B5 40 86 90 E9 4E CD 41 63 95 D0 21
01 42 85 40 2D 41 33 8F 20 00 13 03 0F 00 05 02
89 42 E3 18 52 FE E1 4E D1 41 63 16 D3 1F 01 42
B9 40 2D 41 33 8F 20 00 01 00 13 03 0F 00 05 02
```


图 17-4 Verilog 的 readmemh 函数可读入文件内容片段

用户如果修改了汇编程序的源代码需要重新编译，遵循以下步骤，编译出的文件将被重新生成在 e200_opensource/ riscv-tools/ riscv-tests/isa/generated 目录中。

```
// 注意：下列步骤的完整描述也被记载于 e200_opensource 项目的 doc 目录中的
SoC_Quick_Start_Guide 文档，以便于读者直接复制进行重现。

// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置。
         （1）使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。
         （2）由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统。
         有关如何安装 VMware 以及 Ubuntu 操作系统本书不做介绍，有关 Linux 的基本使用本书
         也不做介绍，请读者自行查阅资料学习。
```

// 步骤二：为了防止后续步骤中出现错误，预先最好将很多工具包先安装在 **Ubuntu 16.04** 系统中，使用如下命令。

```
sudo apt-get install autoconf automake autotools-dev curl device-tree-compiler
    libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo
    gperf libtool patchutils bc zlib1g-dev
```

// 步骤三：将 **e200_opensource** 项目下载到本机 **Linux** 环境中，使用如下命令。

```
git clone https://github.com/SI-RISCV/e200_opensource.git
        // 经过此步骤将项目克隆下来，本机上即可具有如第 17.1 节中所述完整的
        // e200_opensource 目录文件夹，假设该目录为<your_e200_dir>，后文将使用该缩
        // 写指代。
```

// 步骤四：由于编译汇编程序需要使用到 **GNU** 工具链，假设使用完整的 **riscv-tools** 来自己编译 **GNU** 工具链则费时费力，因此本书推荐使用预先已经编译好的 **GCC** 工具链。作者已经将工具链上传至网盘，网盘具体地址记载于 **e200_opensource** 项目 **prebuilt_tools** 目录的 **README** 中。用户可以在网盘中的“**RISC-V Software Tools/RISC-V_GCC_201801_Linux**”目录下载压缩包 **gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz**，然后按照如下步骤解压使用（注意：上述链接网盘上的工具链可能会不断更新，读者请注意自行判断使用最新日期的版本，下列步骤仅为特定版本的示例）。

```
cp gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz ~/
       // 将压缩包复制到用户的根目录下

cd ~/
tar -xzvf gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz
       // 进入根目录并解压该压缩包，解压后可以看到一个生成的 gnu-mcu-eclipse 文件夹

cd <your_e200_dir>/
       // 进入到 e200_opensource 的目录文件夹

mkdir -p ./riscv-tools/prebuilt_tools/prefix/bin
       // 在 e200_opensource 目录下创建上述这个 bin 目录

cd ./riscv-tools/prebuilt_tools/prefix/bin/
       // 进入到这个新建的 bin 目录下

ln -s ~/gnu-mcu-eclipse/riscv-none-gcc/7.2.0-2-20180111-2230/bin/* .
       // 将用户根目录下解压压缩包中 bin 目录下的所有可执行文件作为软链接链接到
       // 该./riscv-tools/prebuilt_tools/prefix/bin/目录下
```

// 步骤五：在接下来的步骤中可能会出现如下错误。
        "Syntax error:Bad fd number"
// 这个错误可能是由于在 **Ubuntu 16.04** 中，**/bin/sh** 被链接到了 **/bin/dash** 而不

```
// 是 /bin/bash. 如果果真如此，可以用以下命令进行修改。
      sudo mv /bin/sh /bin/sh.orig
      sudo ln -s /bin/bash /bin/sh


// 步骤六：使用如下命令编译出 Spike（指令模拟器）和 riscv-tests。

cd <your_e200_dir>/riscv-tools
        // 进入到 e200_opensource 目录下的 riscv-tools 文件夹
./build-e200-spike-rvtests.sh
        // 运行该脚本将编译出指令模拟器 Spike 和 riscv-tests。
        // 如果运行该步骤没有出现错误，那么一个可执行文件 spike 将被生成在
        // <your_e200_dir>/riscv-tools/prebuilt_tools/prefix/bin/ 目录下，
        // 一些相关的库文件也
        // 将被生成在<your_e200_dir>/riscv-tools/prebuilt_tools/prefix/
        // 目录下。
        // 可以通过在<your_e200_dir>/riscv-tools/prebuilt_tools/prefix/bin/
        // 目录下运行 "./spike -h" 来确认此 spike 是否能够被正确执行。


// 步骤七：在以下文件目录下可运行以下命令，编译出的文件将被重新生成在 e200_opensource/
riscv-tools/riscv-tests/isa/generated 目录中。
  e200_opensource
       |----riscv-tools
            |----riscv-tests
                 |----isa              // 在该目录下运行命令 source regen.sh
                      |---- regen.sh
```

注意：如果用户没有修改任何的汇编 test 的源代码，直接运行此"source regen.sh"时，Makefile 认为没有更新，什么都不用做（显示"make: Nothing to be done for default"）。如果用户修改了代码，假设用户修改了上文中提到的 isa/rv64ui/add.S 汇编测试代码（必须同时也修改 isa/rv32ui/add.S 代码，在其中随便添加一个空格，否则 Makefile 的依赖关系无法追踪间接 include 的源代码改动），那么运行"source regen.sh"后，在 generated 目录下的相关 rv32ui-p-addi*文件将会被重新生成。

17.3 E200 开源项目的测试平台（TestBench）

在 e200_opensource 的如下目录已经创建了一个简单的由 Verilog 编写的 TestBench 测试平台。

```
e200_opensource
     |----tb                      // 存放 Verilog TestBench （测试平台）的目录
          |----tb_top.v           // 简单的 Verilog TestBench 顶层文件
```

在测试平台中主要的功能如下。

- 例化 DUT 文件，生成 clock 和 reset 信号。

- 根据运行命令解析出测试用例的名称，并使用 Verilog 的 readmemh 函数读入相应的文件（譬如 rv32ui-p-addi.verilog）内容，然后使用文件中的内容初始化 ITCM（由 Verilog 编写的二维数组充当行为模型），如图 17-5 所示。

```
integer i;

  reg [7:0] itcm_mem [0:(`E200_ITCM_RAM_DP*8)-1];
  initial begin
    $readmemh({testcase, ".verilog"}, itcm_mem);

    for (i=0;i<(`E200_ITCM_RAM_DP);i=i+1) begin
        `ITCM.mem_r[i][00+7:00] = itcm_mem[i*8+0];
        `ITCM.mem_r[i][08+7:08] = itcm_mem[i*8+1];
        `ITCM.mem_r[i][16+7:16] = itcm_mem[i*8+2];
        `ITCM.mem_r[i][24+7:24] = itcm_mem[i*8+3];
        `ITCM.mem_r[i][32+7:32] = itcm_mem[i*8+4];
        `ITCM.mem_r[i][40+7:40] = itcm_mem[i*8+5];
        `ITCM.mem_r[i][48+7:48] = itcm_mem[i*8+6];
        `ITCM.mem_r[i][56+7:56] = itcm_mem[i*8+7];
    end
```

图 17-5 使用 Verilog 的 readmemh 函数读入文件初始化 ITCM

- 在运行结束后分析该测试用例是否执行成功，在 Testbench 的源文件中对 x3 寄存器的值进行判断，如果 x3 的值为 1，则意味着通过，向终端上将打印 PASS 字样，否则将打印 FAIL 字样，如图 17-6 所示。

```
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~ Test Result Summary ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~TESTCASE: %s ~~~~~~~~~~~~~", testcase);
    $display("~~~~~~~~~~~~~~~Total cycle count value: %d ~~~~~~~~~~~~~", cycle_count);
    $display("~~~~~~~~~~The valid Instruction Count: %d ~~~~~~~~~~~~~", valid_ir_cycle);
    $display("~~~~~The test ending reached at cycle: %d ~~~~~~~~~~~~~", pc_write_to_host_cycle);
    $display("~~~~~~~~~~~~~~~~The final x3 Reg value: %d ~~~~~~~~~~~~~", x3);
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
if (x3 == 1) begin
    $display("~~~~~~~~~~~~~~~~ TEST_PASS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~ #####     ##     ####    ####  ~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~ #    #   #  #   #       #      ~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~ #    #  #    #   ####    ####  ~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~ #####   ######       #       #~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~ #       #    #  #    #  #    #~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~ #       #    #   ####    ####  ~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
end
else begin
    $display("~~~~~~~~~~~~~~~~ TEST_FAIL ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~######    ##       #    #     ~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#        #  #      #    #     ~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#####   #    #     #    #     ~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#       ######     #    #     ~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#       #    #     #    #     ~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#       #    #     #    ######~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
```

图 17-6 Testbench 中打印测试用例的结果

17.4 在 Verilog TestBench 中运行测试用例

感兴趣的读者若希望能够运行仿真测试程序，可以使用如下步骤进行。

```
// 注意：下列步骤的完整描述也被记载于 e200_opensource 项目的 doc 目录中的 SoC_Quick_Start_
Guide 文档，以便于读者直接复制进行重现。

// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置。
          （1）使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。
          （2）由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统
          有关如何安装 VMware 和 Ubuntu 操作系统本书不做介绍，有关 Linux 的基本使用本书
          也不做介绍，请读者自行查阅资料学习。
```

```
// 步骤二：将 e200_opensource 项目下载到本机 Linux 环境中，使用如下命令。

git clone https://github.com/SI-RISCV/e200_opensource.git
        // 经过此步骤将项目克隆下来，本机上即可具有如第 17.1 节中所述完整的
        // e200_opensource 目录文件夹，假设该目录为<your_e200_dir>，后文将使用该缩
        // 写指代。

// 步骤三：编译 RTL 代码，使用如下命令。

cd <your_e200_dir>/vsim
        // 进入到 e200_opensource 目录文件夹下面的 vsim 目录。

make install CORE=e203
        // 运行该命令指明需要为 e203 进行编译，该命令会在 vsim 目录下生成一个 install
        // 子文件夹，在其中放置所需的脚本，且将脚本中的关键字设置为 e203。

make compile
        // 编译 Core 和 SoC 的 RTL 代码
        // 注意：在此步骤之中，编译 Verilog 代码需要使用到仿真器工具，在 E200 的 Makefile
        // 中使用的是开源免费的 iverilog 工具，如图 17-7 所示。
        // 对于开源免费的 iverilog 工具如何安装请读者在互联网上自行搜索。

// 步骤四：运行默认的一个 testcase（测试用例），使用如下命令。
make run_test
        // 注意：在此步骤中，运行仿真需要使用仿真器工具，在开源的 Makefile 中此部分空缺，
        // 实际运行的是"echo PASS"命令打印一个虚假的 PASS 到 log 文件中，如图 17-7 所示。
        //
        // 注意：make run_test 将运行 e200_opensource/riscv-tools/
        // riscv-tests/isa/generated 目录中的一个默认 testcase，如果希望运行所有的
        // 回归测试，请参见步骤五。

// 步骤五：运行回归（regression）测试集，使用如下命令。
make regress_run CORE=e203
        // 该命令使用 e200_opensource/riscv-tools/riscv-tests/isa/generated
        // 目录中 E203 Core 的 testcases，逐个的运行各个 testcase。

// 步骤六：查看回归测试结果。
make regress_collect CORE=e203
        // 该命令将收集步骤五中运行的测试集的结果，将打印若干行的结果，每一行对应一个测
        // 试用例，如果那个测试用例运行通过，那一行则打印的 PASS；如果运行失败，那一行则
        // 打印的 FAIL，如图 17-8 所示。
        //
```

```
RTL_V_FILES      := $(wildcard ${VSRC_DIR}/*/*.v)
TB_V_FILES       := $(wildcard ${VTB_DIR}/*.v)


SIM_TOOL       := #To-ADD: to add the simulatoin tool
SIM_TOOL       := iverilog # this is a free solution here to use iverilog to compile the code

SIM_OPTIONS    := #To-ADD: to add the simulatoin tool options


SIM_EXEC       := #To-ADD: to add the simulatoin executable
SIM_EXEC       := echo "Test Result Summary: PASS" # This is a fake run to just direct print PASS
```

图 17-7 仿真工具的设置

```
PASS ./run/rv32um-p-rem/rv32um-p-rem.log
PASS ./run/rv32um-p-mulh/rv32um-p-mulh.log
PASS ./run/rv32ua-p-amomax_w/rv32ua-p-amomax_w.log
PASS ./run/rv32ua-p-amominu_w/rv32ua-p-amominu_w.log
PASS ./run/rv32ua-p-amoor_w/rv32ua-p-amoor_w.log
PASS ./run/rv32ui-p-simple/rv32ui-p-simple.log
PASS ./run/rv32ui-p-beq/rv32ui-p-beq.log
PASS ./run/rv32mi-p-scall/rv32mi-p-scall.log
PASS ./run/rv32ui-p-lbu/rv32ui-p-lbu.log
PASS ./run/rv32ui-p-jal/rv32ui-p-jal.log
PASS ./run/rv32ui-p-fence_i/rv32ui-p-fence_i.log
PASS ./run/rv32ua-p-amoxor_w/rv32ua-p-amoxor_w.log
PASS ./run/rv32ui-p-bge/rv32ui-p-bge.log
PASS ./run/rv32ui-p-ori/rv32ui-p-ori.log
PASS ./run/rv32mi-p-breakpoint/rv32mi-p-breakpoint.log
PASS ./run/rv32ui-p-sh/rv32ui-p-sh.log
PASS ./run/rv32mi-p-mcsr/rv32mi-p-mcsr.log
PASS ./run/rv32ui-p-sll/rv32ui-p-sll.log
PASS ./run/rv32um-p-mulhu/rv32um-p-mulhu.log
PASS ./run/rv32uc-p-rvc/rv32uc-p-rvc.log
PASS ./run/rv32ui-p-srli/rv32ui-p-srli.log
PASS ./run/rv32ui-p-sra/rv32ui-p-sra.log
PASS ./run/rv32um-p-remu/rv32um-p-remu.log
PASS ./run/rv32ui-p-lb/rv32ui-p-lb.log
PASS ./run/rv32ui-p-or/rv32ui-p-or.log
PASS ./run/rv32ui-p-slli/rv32ui-p-slli.log
PASS ./run/rv32ua-p-amomaxu_w/rv32ua-p-amomaxu_w.log
```

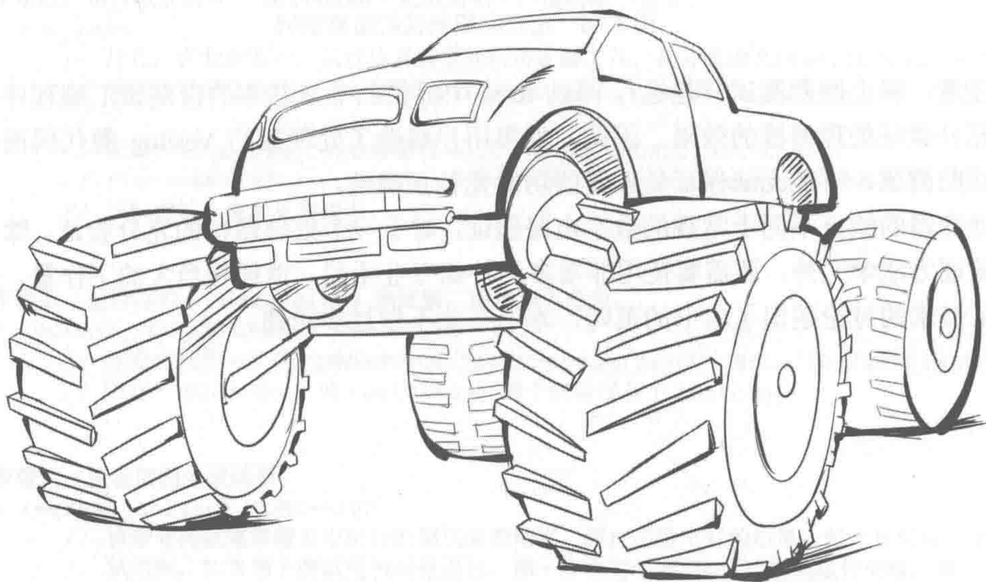
图 17-8 运行回归测试的结果示例

注意：以上回归测试只是运行 riscv-tests 中提供的非常基本的自测试汇编程序，并不能达到充分验证处理器核的效果。因此，如果用户修改了处理器的 Verilog 源代码而仅运行以上的回归测试，可能无法保证处理器的功能完备正确性。

处理器的验证不同于常规的数字电路验证，对于一个处理器核的充分验证，除了使用常规的验证方法学之外，还需要使用非常多的特殊专业手段，消耗相当大的工作量。关于处理器验证技术的讨论超出了本书的范畴，本书在此不做过多论述。

第 18 章　套上壳子上路——实现 SoC 和 FPGA 原型

套上帅帅的壳子



仅仅一个处理器核无法真正运行，就像一辆汽车只有发动机是无法行驶的，需要给它套上壳子、安上轮子才能上路。对于一个处理器核，还需要配套 SoC 才能具备完整的功能。本章将介绍一款开源的 Freedom E310 SoC，并介绍蜂鸟 E200 处理器配套的 SoC，以及如何在 FPGA 上实现该 SoC 原型。

18.1 Freedom E310 SoC 简介

在介绍 Freedom E310 SoC 平台之前，在此先介绍 Freedom E300 平台。

Freedom E300 平台由 SiFive 公司推出，SiFive 公司是由美国加州大学伯克利分校发明 RISC-V 架构的几个主要发起人创办的商业公司，力图加速 RISC-V 的商业化进程与生态的推广。

SiFive 公司目前已经发布了几款 RISC-V 架构的商用处理器核 IP，还发布了几款 SoC 平台系列。其中，Freedom Everywhere 是一款可配置的 RISC-V SoC 家族系列，主要面向低功耗的嵌入式 MCU 领域。而 E300 平台是 Freedom Everywhere SoC 家族系列中的第一款 SoC 平台，关于 Freedom Everywhere E300 平台的具体信息，请在 SiFive 的官方网址中（无须注册）下载其技术手册“SiFive-E300-platform-reference-manual.pdf”。

Freedom Everywhere E310-G000（简称 Freedom E310）是使用 Freedom Everywhere E300 平台配置出的一款特定配置 SoC，SiFive 将此 SoC 的代码完全开源，读者可以在 GitHub 中搜索“freedom”来下载其源代码。Freedom E310 SoC 基于 Rocket Core，架构配置为 RV32IMAC 架构，配备 16KB 的指令 Cache 与 16KB 的数据 SRAM（Scratchpad）、硬件乘除法器、调试（Debug）模块，以及丰富的外设，如 PWM、UART 和 SPI 等，其结构框图如图 18-1 所示。

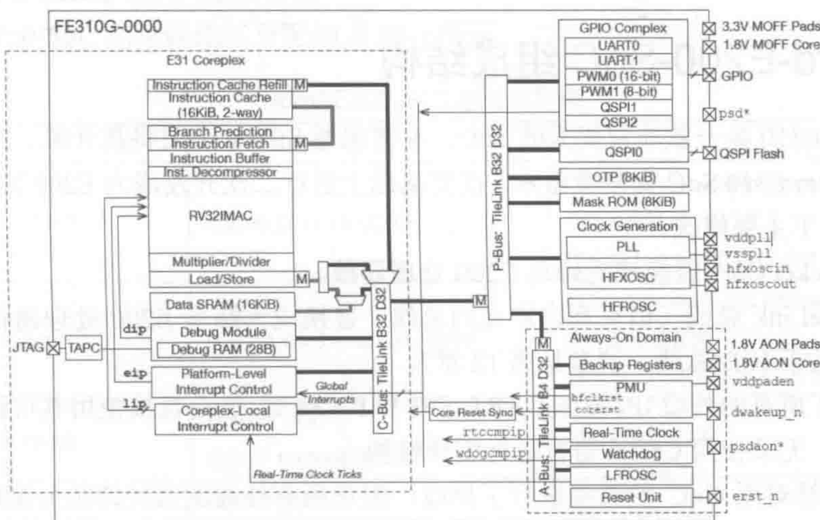


图 18-1 Freedom 310 SoC 结构图

基于 Freedom E310 SoC，SiFive 公司使用 TSMC CL018G 180nm 工艺成功流片（Tapeout）生产出实际芯片，能够运行到 320MHz 以上的主频。基于此芯片，SiFive 公司开发制造了一款信用卡大小的硬件开发板 HiFive1，如图 18-2 所示。HiFive1 开发板是目前市场上最早的在售 RISC-V 硬件开发板，已经被很多 RISC-V 的公司与爱好者使用。

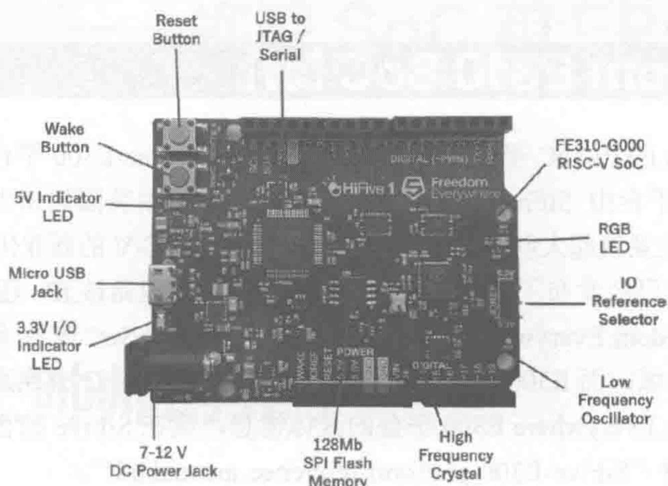


图 18-2 HiFive1 开发板结构图

18.2 HBird-E200-SoC 简介

18.2.1 HBird-E200-SoC 组成结构

Freedom E310 是一款非常优秀的 SoC，非常感谢 SiFive 公司将其开源。E200 开源项目也是以 Freedom E310 SoC 为参考蓝本，在其基础上进行二次开发成为 E200 配套的 SoC，其中主要做了如下主要修改。

- 将 Rocket Core 替换成为蜂鸟 E200 处理器核。
- 将 TileLink 总线（伯克利自定义的总线）替换成为蜂鸟 E200 处理器使用的 ICB 总线（关于 ICB 总线，请参见第 12 章）。
- 保留了所有的外设 IP，譬如 UART、SPI 和 PWM 等，但是直接使用其可综合的 Verilog 代码，无须使用 Chisel 语言进行编译转换。

注意：虽然对原 SoC 的总线进行了修改，但是所有外设的总线地址分配表仍然完全与原始的 Freedom E310 SoC 一致，如表 18-1 所示。因此从软件的角度来看，可以认为修改后

的 SoC（虽然使用的是蜂鸟 E200 处理器核）与原始的 Freedom E310 几乎完全兼容，软件可以很方便移植运行。

为了方便读者理解区别，本书将“修改后的 SoC（E200 处理器核开源配套的 SoC）”称为“HBird-E200-SoC”，同时非常感谢 SiFive 公司开源 Freedom SoC 为 RISC-V 社区所做的贡献。

1. HBird-E200-SoC 结构图

HBird-E200-SoC 结构如图 18-3 所示。

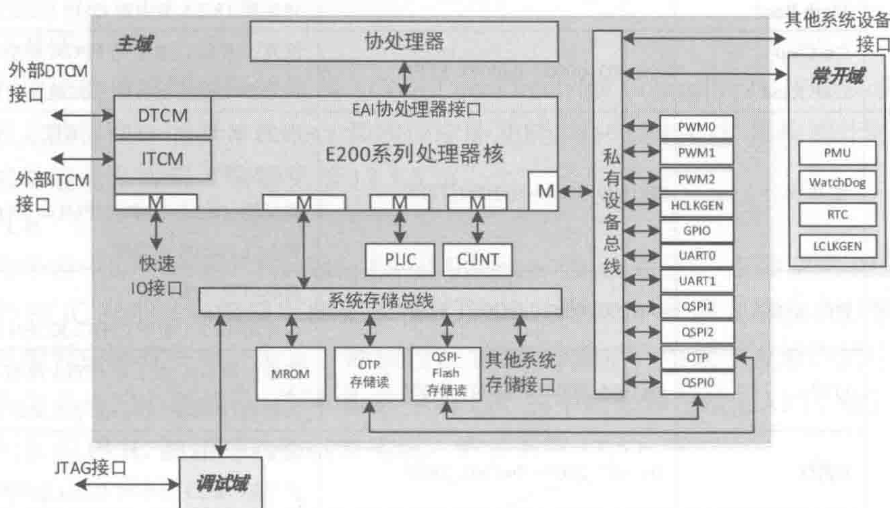


图 18-3　HBird-E200-SoC 结构图

2. HBird-E200-SoC 地址分配

HBird-E200-SoC 的总线地址分配如表 18-1 所示。

表 18-1　HBird-E200-SoC 地址分配表

总线分组	组　件	地址区间	描　述
处理器核直属	CLINT	0x0200_0000 ~ 0x0200_FFFF	CLINT（Core Local Interrupt Controller）模块寄存器地址区间 详见第 18.2.1 节中对 CLINT 的介绍
	PLIC	0x0C00_0000 ~ 0x0CFF_FFFF	PLIC（Platform Level Interrupt Controller）模块寄存器地址区间 详见第 18.2.1 节中对 PLIC 的介绍
	ITCM	0x8000_0000 ~ 取决于 ITCM 配置大小	ITCM 地址区间
	DTCM	0x9000_0000 ~ 取决于 DTCM 配置大小	DTCM 地址区间

续表

总线分组	组　件	地址区间	描　述
系统存储总线接口	调试模块	0x0000_0000 ~ 0x0000_0FFF	注意：调试模块主要用于调试器使用，普通软件程序不应该使用此区间
	Mask-ROM	0x0000_1000 ~ 0x0000_1FFF	注意：开源的 E200 项目由于是 FPGA 原型，因此 Mask-ROM 代码为一行为模型
	Off-Chip QSPI0 Flash Read	0x2000_0000 ~ 0x3FFF_FFFF	外部 SPI Flash 只读地址区间 详见第 18.2.1 节中对 QSPI Flash 的介绍
	On-Chip OTP Read	0x0002_0000 ~ 0x0003_FFFF	注意：本 SoC 由于是 FPGA 原型，并未实际提供 OTP 模块，仅分配了此地址且连接一空模块
私有外设总线接口	常开域	0x1000_0000 ~ 0x1000_7FFF	常开域包含 PMU、RTC、WatchDog 和 LCLKGEN 详见第 18.2.1 节中对 PMU、RTC、WatchDog 和 LCLKGEN 的介绍
	HCLKGEN	0x1000_8000 ~ 0x1000_8FFF	高速时钟生成模块 详见第 18.2.1 节中对 HCLKGEN 的介绍
	OTP	0x1001_0000 ~ 0x1001_0FFF	注意：本 SoC 由于是 FPGA 原型，并未实际提供此 OTP 模块，仅分配了此地址连接一空模块
	GPIO	0x1001_2000 ~ 0x1001_2FFF	GPIO 地址区间 详见第 18.2.1 节中对 GPIO 的介绍
	UART0	0x1001_3000 ~ 0x1001_3FFF	第一个 UART 模块地址区间 详见第 18.2.1 节中对 UART 的介绍
	QSPI0	0x1001_4000 ~ 0x1001_4FFF	第一个 QSPI 模块地址区间 详见第 18.2.1 节中对 QSPI 的介绍
	PWM0	0x1001_5000 ~ 0x1001_5FFF	第一个 PWM 模块地址区间 详见第 18.2.1 节中对 PWM 的介绍
	UART1	0x1002_3000 ~ 0x1002_3FFF	第二个 UART 模块地址区间
	QSPI1	0x1002_4000 ~ 0x1002_4FFF	第二个 QSPI 模块地址区间
	PWM1	0x1002_5000 ~ 0x1002_5FFF	第二个 PWM 模块地址区间
	QSPI2	0x1003_4000 ~ 0x1003_4FFF	第三个 QSPI 模块地址区间
	PWM2	0x1003_5000 ~ 0x1003_5FFF	第三个 PWM 模块地址区间
其他地址区间	上表中未使用到的地址区间，则均为写忽略，读返回 0		

3. HBird-E200-SoC 模块简述

本书限于篇幅，仅对每个模块功能进行简略的描述。感兴趣的读者可到 e200_opensource 项目的 doc 目录中查看此 SoC 的详细设计文档。

HBird-E200-SoC 中主要的模块简述如下。

（1）HCLKGEN

- 整个 SoC 从时钟域（Clock Domains）上分为两个大的部分：主体域（Main Domain）和电源常开域（Always-on Domain）。其中，主体域使用高速的时钟，而常开域使用低速的时钟。
- HCLKGEN 全称为 High-Speed Clock Generation（高速时钟生成），该模块主要为主体域生成高速时钟。高速时钟来自于外部 16MHz 晶振产生或者内部的 PLL 产生的高频时钟。

（2）CLINT

全称为处理器核局部中断控制器（Core-Local Interrupt Controller），主要实现 RISC-V 架构手册中规定的标准计时器（Timer）和软件中断功能。有关 RISC-V 架构的计时器中断和软件中断详细功能以及实现，请参见第 13.5.5 节。

（3）PLIC

全称为平台中断控制器（Platform-Level Interrupt Controller），主要实现 RISC-V 架构手册中规定的 PLIC 功能（详细信息参见本书附录 C）。该 PLIC 能够支持多个中断源，并且每个中断可以配置中断优先级，所有中断源经过 PLIC 仲裁后，生成一根最终的中断信号通给处理器核作为其外部中断信号。在本 SoC 中，PLIC 的中断来源包括 UART、SPI 和 GPIO 等，其中断分配表和 PLIC 的详细功能以及实现，请参见第 13.5.6 节。

（4）JTAG

标准（1149.1）JTAG 连接模块用于连接系统外部调试器（Debugger）与内部的调试模块（Debug Module）。

（5）调试模块

调试模块，用于支持外部 JTAG 通过该模块调试处理器核，使得处理器核能够通过 GDB 对其进行交互式调试，譬如设置断点、单步执行等调试功能。

（6）Quad-SPI Flash

- 专用于连接外部 Flash 的 Quad-SPI（QSPI）接口。
- 该 QSPI 接口可以被软件配置成为 eXecute-In-Place 模式，在此模式下，Flash 可以被当作一段只读区间直接被存储器读取。在默认上电之后，QSPI 即处于该模式之下。由于 Flash 掉电不丢失的特性，因此可以将系统的启动程序存放于外部的 Flash 中，然后处理器核通过 eXecute-In-Place 模式的 QSPI 接口直接访问外部 Flash，加载启动程序启动。

（7）GPIO

- 全称为 General Purpose I/O，用于提供一组 32 I/O 的通用输入输出接口。每个 I/O 可被软件配置为输入或者输出，如果是输出，则可以设置具体的输出值。

- 每个 I/O 还可以被配置为 IOF（Hardware I/O Functions），也就是将 I/O 供 SoC 内部的其他模块复用，譬如 SPI、UART 和 PWM 等。
- GPIO 的 32 个 I/O 被 SoC 内部模块的复用分配如表 18-2 所示，其中每个 I/O 均可以供两个内部模块复用，软件可以通过配置每个 I/O 使其选择 IOF0 或者 IOF1 来选择信号来源。
- 另外，每个 GPIO 的 I/O 均作为一个中断源被连接到 PLIC 的中断源上。

表 18-2　GPIO 的接口分配

GPIO Pin 编号	IOF0	IOF1
0	—	PWM0_0
1	—	PWM0_1
2	QSPI1:SS0	PWM0_2
3	QSPI1:SD0/MOSI	PWM0_3
4	QSPI1:SD1/MISO	—
5	QSPI1:SCK	—
6	QSPI1:SD2	—
7	QSPI1:SD3	—
8	QSPI1:SS1	—
9	QSPI1:SS2	—
10	QSPI1:SS3	PWM2_0
11	—	PWM2_1
12	—	PWM2_2
13	—	PWM2_3
14	—	—
15	—	—
16	UART0:RX	—
17	UART0:TX	—
18	—	—
19	—	PWM1_1
20	—	PWM1_0
21	—	PWM1_2
22	—	PWM1_3
23	—	—
24	UART1:RX	—
25	UART1:TX	—
26	QSPI2:SS	—
27	QSPI2:SD0/MOSI	—

续表

GPIO Pin 编号	IOF0	IOF1
28	QSPI2:SD1/MISO	—
29	QSPI2:SCK	—
30	QSPI2:SD2	—
31	QSPI2:SD3	—

（8）QSPI

除了上述专用于 Flash 的 QSPI 接口之外，SoC 还有两个独立的 QSPI 接口控制器。一个 QSPI 使用 4 个片选信号（Chip Selects），1 个 QSPI 使用 1 个片选信号。两个 QSPI 均使用 GPIO 的 IOF 功能与外界通信。

（9）UART

全称为 Universal Asynchronous Receiver-Transmitter（通用异步接收-发射器）。SoC 有两个独立的 UART，两个 UART 均使用 GPIO 的 IOF 功能与外界通信。

（10）PWM

全称为 Pulse-Width Modulator（脉宽调节器）。SoC 有 3 个独立的 PWM，其中 PWM1 和 PWM2 是 16 比特宽的精度，另一个 PWM0 是 8 比特宽的精度。3 个 PWM 均使用 GPIO 的 IOF 功能与外界通信。

（11）常开域

- LCLKGEN：全称为低速时钟生成（Low-Speed Clock Generation）。该模块主要为常开域生成低速时钟，应该使用 32.768KHz 的低速实时时钟，该时钟来自于外部晶振产生。
- WatchDog：全称为看门狗计数器（Watch Dog Timer）。该计数器位于常开域中，因此使用低速时钟进行计数，并且可以通过配置其计数的目标值产生中断。
- RTC：全称为实时计数器（Real-Time Counter）。该计数器位于常开域中，因此使用低速时钟进行计数，并且还能产生中断。
- PMU：全称为电源管理单元（Power Management Unit），用于控制 SoC 的电源管理。整个 SoC 除了 WatchDog、RTC 和 PMU 等模块处于常开域之外，其他主域可以在 PMU 的控制下被置于断电状态以节省功耗，或者重新唤醒等。

18.2.2 HBird-E200-SoC 代码结构

HBird-E200-SoC 的代码结构如下所示。

```
e200_opensource
    |----rtl                        // 存放 RTL 的目录
        |----e203                   // E203 核和 SoC 的 RTL 目录
```

```
        |----general      // 存放一些公用的通用 RTL 代码
        |----core         // 存放e203 Core 的 RTL 代码
        |----fab          // 存放总线结构（bus fabric）的 RTL 代码
             |---- sirv_icb1to4_bus.v // 将 1 组 ICB 总线转换成 4 路 ICB 总线
             |---- sirv_icb1to8_bus.v // 将 1 组 ICB 总线转换成 8 路 ICB 总线
             |---- sirv_icb1to16_bus.v // 将 1 组 ICB 总线转换成 16 路 ICB 总线
        |----subsys         // 存放完整子系统顶层的 RTL 代码
             |---- e203_subsys_top.v       // 子系统的顶层
             |---- e203_subsys_main.v      // 子系统的主体部分（可关电）顶层
             |---- e203_subsys_plic.v      // PLIC 顶层
             |---- e203_subsys_clint.v     // CLINT 顶层
             |---- e203_subsys_mems.v      // 子系统的存储部分顶层
             |---- e203_subsys_perips.v    // 子系统的外设部分顶层
        |----mems             // 存放存储器模块的 RTL 代码
        |----perips         // 存放外设（peripherals）模块的 RTL 代码
             |---- sirv_aon_*.v            //常开域部分模块
             |---- sirv_clint*.v           // CLINT 的模块
             |---- sirv_flash_qspi*.v      // Flash 专用的 QSPI 模块
             |---- sirv_gpio*.v            // GPIO 的模块
             |---- sirv_plic*.v            // PLIC 的模块
             |---- sirv_pmu*.v             // PMU 的模块
             |---- sirv_pwm16*.v           // 16bits 精度的 PWM 模块
             |---- sirv_pwm8*.v            // 8bits 精度的 PWM 模块
             |---- sirv_qspi_1cs*.v        // 1 个 CS 选通的 QSPI 模块
             |---- sirv_qspi_4cs*.v        // 4 个 CS 选通的 QSPI 模块
             |---- sirv_qspi*.v            // 其他 QSPI 子模块
             |---- sirv_rtc*.v             // RTC 模块
             |---- sirv_uart*.v            // UART 模块
             |---- sirv_wdog*.v            // WatchDog 模块
        |----debug          // 存放调试相关模块的 RTL 代码
        |----soc          // 存放 soc 顶层的 RTL 代码
             |----e203_soc_top.v          // SoC 顶层
```

各个主要的代码模块简述如下。

（1）general 目录主要用于存放一些通用的 Verilog RTL 模块供整个 SoC 公用，譬如一些 DFF（D 触发器寄存器）模块文件、ICB 总线的基础模块等。

（2）core 目录主要用于存放处理器核模块的 Verilog RTL 代码，关于此部分的具体代码分析，请参见第 5.5 节详述。

（3）fab 目录主要实现 SoC 中 ICB 总线结构模块的 Verilog RTL 代码。sirv_icb1to4_bus.v、sirv_icb1to8_bus.v 或者 sirv_icb1to16_bus.v 实际例化调用了"ICB 分发"模块将一组 ICB 总线按照地址区间分发成为 4 组、8 组或者 16 组 ICB 总线，"ICB 分发"模块的微架构在第 12.3.1 节有详细论述，本书于此不再赘述。

（4）subsys 目录包含了 SoC 的主体顶层模块的 Verilog RTL 代码，其中 e203_subsys_top 是事实上的 SoC 顶层文件，它例化了主域模块（e203_subsys_main.v）、常开域模块（e203_aon_top.v）

和调试模块（sirv_debug_module）。其模块间的层次结构如图 18-4 所示。

（5）mems 目录主要用于存放存储器模块的 Verilog RTL 代码，由于存储器的具体实现依赖于芯片生产加工厂（Foundry），譬如 SMIC 或者 TSMC 的存储器单元，因此本文件夹下的 Verilog RTL 代码仅是行为模型。

（6）perips 目录主要用于存放各种外设（Peripherals）模块的 Verilog RTL 代码，譬如 GPIO、UART 和 SPI 等。

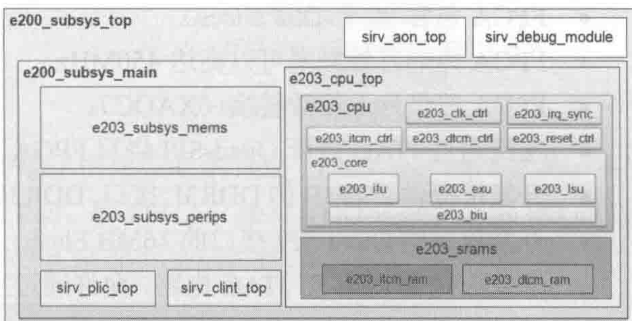


图 18-4　SoC 模块层次结构图

（7）debug 目录包含了 SoC 中有关调试模块的 Verilog RTL 代码。该部分已经在第 14 章详细讨论过，在此不再赘述。

（8）soc 目录主要用于存放 SoC 顶层模块的 Verilog RTL 代码。

- e203_soc_top.v 是一个简单的顶层 SoC 顶层模块，将 e203_subsys_top 进行例化。

18.3 HBird-E200-SoC FPGA 原型平台

E200 开源项目使用 18.2 节中介绍的 HBird-E200-SoC 搭建了完整的 FPGA 原型平台与示例。

FPGA 原型主要分为两部分：FPGA 开发板和调试器。后续章节将分别予以介绍。完整的 FPGA 开发板原型（包括 FPGA 开发板和调试器）如图 18-5 所示。



图 18-5　FPGA 开发板原型（包括调试器）

18.3.1 FPGA 开发板

FPGA 开发板使用 Xilinx Artix-7 35T Arty FPGA Evaluation Kit 开发板。如图 18-6 所示，该开发板是一款低成本的入门级 Xilinx FPGA 开发板，特别适用于电路设计规模不大的 FPGA 爱好者，其特性如下。

- 使用 Xilinx Artix-35T FPGA (xc7a35ticsg324-1L) 。
- FPGA 包含 33280 个逻辑单元，由 5200 个 Slices 组成。每个 Slice 包含 4 个 6 输入的 LUT 和 8 个寄存器。
- FPGA 包含 1800Kbits 的快速 Block RAM。

- FPGA 包含 5 个时钟管理堆，每个均配备 PLL。
- FPGA 包含 90 个 DSP slices。
- FPGA 内部时钟频率可以高达 450MHz。
- FPGA 提供片上模数转换（XADC）。
- 支持通过 JTAG 或者 Quad-SPI 烧写 FPGA。
- 开发板提供 256MB 的 DDR3L 接口，DDR3L 的数据宽度为 16 比特，频率为 667MHz。
- 开发板提供 Quad-SPI 接口的 16MB Flash。
- 开发板包含 USB 转 JTAG 电路，允许通过主机 PC 的 USB 口对 FPGA 的 JTAG 口进行烧录 FPGA。
- 开发板支持通过 USB 接口供电或者通过单独的电源供电。
- 开发板提供 10/100 Mbps 的以太网接口。
- 开发板提供 UART 转 USB 电路，允许 FPGA 通过 UART 接口转 USB 接口与主机 PC 通信。
- 开发板提供 4 个通用按键、1 个 Reset 按键、4 个 LED 灯、4 个 RGB LED 灯。
- 开发板提供 4 个 Pmod connectors 和 1 组 Arduino/chipKIT Shield connector。

注意：该 FPGA 开发板可以直接使用 USB 进行电源供电，或者单独使用电源线供电。出于简便，示例中直接使用 USB 线供电，使用 USB A to Micro-B Cable（即安卓手机充电器 USB 线）对其进行供电即可，如图 18-7 所示。同时该 USB 线也会被用于将 FPGA 的 bitstream 文件（由 Vivado 软件编译 Verilog 所得）烧录到 FPGA 芯片中。

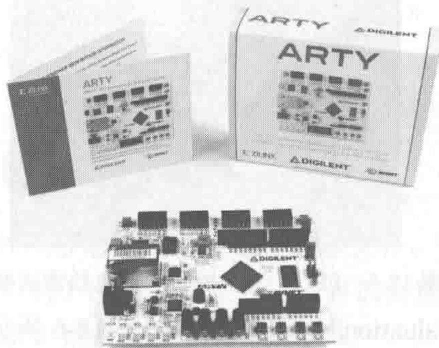


图 18-6 Arty 开发板

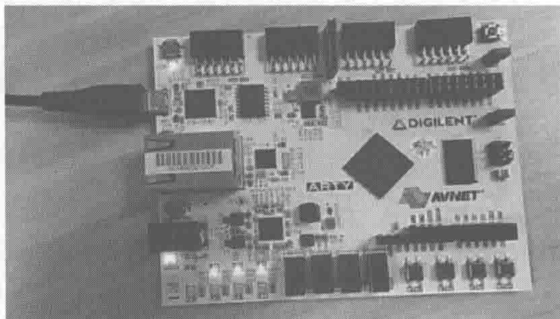


图 18-7 USB A to Micro-B Cable 对 FPGA 开发板供电

E200 开源项目 FPGA 项目相关的代码结构如下所示。

```
e200_opensource
    |----fpga                              // 存放 RTL 的目录
        |----artydevkit                    // Arty 开发板的项目文件夹
            |----constrs                   // 存放约束文件的文件夹
```

```
            |---- arty-master.xdc      // 主约束文件
        |----Makefile                  // Makefile 脚本
        |----script                    // 存放运行脚本的文件夹
        |----src                       // 存放 Verilog 源代码的文件夹
            |----system.org            // FPGA 系统的顶层模块
    |----Makefile                      // Makefile 脚本
```

在 FPGA 的顶层模块（system.org）中除了例化 SoC 的顶层（e203_soc_top）之外，主要是使用 Xilinx 的 I/O Pad 单元例化顶层的 Pad。譬如 JTAG 接口 TDO 连接到名为 jd0 的 Pad 上，如图 18-8 所示，另外使用 Xilinx 的 MMCM 单元生成时钟。

```
IOBUF_jtag_TDI

   O iobuf_jtag_TDI_o
     jd_4
   I dut_io_pads_jtag_TDI_o_oval
   T  dut_io_pads_jtag_TDI_o_oe

      dut_io_pads_jtag_TDI_i_ival   iobuf_jtag_TDI_o   dut_io_pads_jtag_TDI_o_ie
      pullup_TDI   O jd_4

    iobuf_jtag_TDO_o

IOBUF_jtag_TDO

   O iobuf_jtag_TDO_o
     jd_0
   I dut_io_pads_jtag_TDO_o_oval
   T  dut_io_pads_jtag_TDO_o_oe

      dut_io_pads_jtag_TDO_i_ival   iobuf_jtag_TDO_o   dut_io_pads_jtag_TDO_o_ie
```

图 18-8　system 中 I/O Pad 例化

HBird-E200-SoC 的顶层 I/O Pad 经过 FPGA 的约束文件（arty-master.xdc）进行约束，使之连接到 FPGA 芯片外部真实的 pin 脚上面，譬如 JTAG 的 Pad（jd0～jd7）被连接到了 FPGA 芯片的 D4/D3 等 pin 脚上，如图 18-9 所示。

```
set_property -dict { PACKAGE_PIN D4    IOSTANDARD LVCMOS33 } [get_ports { jd_0 }];
set_property -dict { PACKAGE_PIN D3    IOSTANDARD LVCMOS33 } [get_ports { jd_1 }];
set_property -dict { PACKAGE_PIN F4    IOSTANDARD LVCMOS33 } [get_ports { jd_2 }];
set_property -dict { PACKAGE_PIN E2    IOSTANDARD LVCMOS33 } [get_ports { jd_4 }];
set_property -dict { PACKAGE_PIN D2    IOSTANDARD LVCMOS33 } [get_ports { jd_5 }];
set_property -dict { PACKAGE_PIN H2    IOSTANDARD LVCMOS33 } [get_ports { jd_6 }];

set_property -dict { PACKAGE_PIN D10   IOSTANDARD LVCMOS33  } [get_ports { uart_rxd_out }];
set_property -dict { PACKAGE_PIN A9    IOSTANDARD LVCMOS33  } [get_ports { uart_txd_in }];
```

图 18-9　arty-master.xdc 中的 pin 脚约束

18.3.2 生成 mcs 文件烧写 FPGA

在第 18.2 节中介绍了 E200 开源项目的 SoC 整体架构和 Verilog RTL 代码，为了使得该 SoC 能够真正运行在 FPGA 硬件上，需要将其编译成为 bitstream 文件，然后烧录到 FPGA 中去，步骤如下。

```
// 注意：下列步骤的完整描述也被记载于 e200_opensource 项目的 doc 目录中的
SoC_Quick_Start_Guide 文档，以便于读者直接复制命令进行运行。

  // 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行。如果是个人用户，推荐如下配置。
          (1) 使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。
          (2) 由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统
          有关如何安装 VMware 和 Ubuntu 操作系统本书不做介绍，有关 Linux 的基本使用本书
          也不做介绍，请读者自行查阅资料学习。

  // 步骤二：安装 Xilinx Vivado 软件至此虚拟机 Linux 操作系统中。有关如何安装 Xilinx Vivado
  //         软件本书不做介绍，请读者自行查阅资料了解。

  // 步骤三：将 e200_opensource 项目下载到本机 Linux 环境中，使用如下命令。

  git clone https://github.com/SI-RISCV/e200_opensource.git
          // 经过此步骤将项目克隆下来，本机上即可具有如第 17.1 节中所述完整的
          // e200_opensource 目录文件夹，假设该目录为<your_e200_dir>，后文将使用该缩
          // 写指代。

  // 步骤四：设置需要编译的 E200 Core 的具体型号，使用如下命令。

  cd <your_e200_dir>/fpga
          // 进入到 e200_opensource 目录文件夹下面的 fpga 目录。

  make install CORE=e203
          // 运行该命令指明需要为 e203 core 进行编译，该命令会在 fpga 目录下生成一个
          // install 子文件夹，在其中放置 Vivado 所需的脚本，且将脚本中的关键字设置为 e203。
          // 同时在 fpga 的 install/rtl 子目录下将生成 FPGA 项目所需的所有 RTL 文件，包括
          // system.v 和其他 RTL 源文件。
          // 注意：
          //     由于上述"make install CORE=e203"命令通过 fpga/common.mk 脚本将
          //     添加一个特殊的宏 FPGA_SOURCE 至 e203_defines.v。
          //     因此：对于 FPGA 项目，必须使用此 install/rtl 目录下的文件。

  // 步骤五：安装 Arty 开发板的 board part 到 Vivado 软件中。
          // 如果第一次使用 Vivado 软件为 Arty 开发板进行编译，则可能出现如图 18-10 所示的
          // 错误，这是因为没有为 Arty 开发板安装 board part。
          // 按照 Digilent 公司网址链接（请在百度中搜索 digilent arty board-files）中
```

```
        // 的说明安装 Arty 开发板的 board part 即可。

// 步骤六：生成 bitstream 文件或者 mcs 文件（推荐使用 mcs 文件），使用如下命令。

make bit
        // 运行该命令将调用 Vivado 软件对 Verilog RTL 进行编译生成 bitstream 文件。
        // 注意：为了保证此步骤能够成功执行，请使用较新版本的 Vivado；如果是在虚拟机
        // 中运行的 Linux 操作系统，确保为虚拟机分配超过 4G 的内存空间，否则 Vivado
        // 可能因内存不够而中断退出。
        // 生成的 bitstream 文件名和路径为
        // <your_e200_dir>/fpga/artydevkit/obj/system.bit
        // 该 bitstream 文件则可以使用 Vivado 软件的 Hardware Manager 功能将
        // system.bit 烧录至 FPGA 中去。

        // 熟悉 Vivado 和 Xilinx FPGA 使用的用户应该了解，bitstream 文件烧录到 FPGA 中
        // 去之后 FPGA 不能掉电，因为一旦掉电之后，FPGA 烧录的内容即丢失，需要重新使用
        // Vivado 的 Hardware Manager 进行烧录方能使用。
        // 为了方便用户使用，Xilinx 的 Arty 开发板可以将需要烧录的内容写入开发板上的
        // Flash 中，然后在每次 FPGA 上电之后通过硬件电路自动将需要烧录的内容从外部的
        // Flash 中读出并烧录到 FPGA 之中（该过程比较的快，不影响用户使用）。由于 Flash
        // 是非易失性的存储器，具有掉电后仍可保存的特性。这意味着将需要烧录的内容写入
        // Flash 后，每次掉电后无需使用 Hardware Manager 人工重新烧录（而是硬件电路
        // 快速自动完成），即等效于 FPGA 上电即可使用。
        // 有关此特性的详细原理与描述，本书不做赘述，请读者自行参阅 Arty 开发板手册。

        // 为了能够将烧录 FPGA 的内容写入 Flash 中，需要生成 mcs 文件，使用如下命令。
make mcs
        // 运行该命令将调用 Vivado 软件对 Verilog RTL 进行编译生成 mcs 文件
        // 生成的 mcs 文件名和路径为
        // <your_e200_dir>/fpga/artydevkit/obj/system.mcs
        // 该 mcs 文件则可以使用 Vivado 软件的 Hardware Manager 功能将
        // system.mcs 烧录至 FPGA 开发板中的 Flash 中去。
```

图 18-10 未安装 Arty 开发板的 board part 到 Vivado 软件中引起的错误

如何使用 Vivado 的 Hardware Manager 烧写 mcs 文件至 FPGA 开发板上的 Flash 中去，参考如下步骤。

// 步骤一：打开 **Vivado** 软件。

// 步骤二：打开 **Hardware Manager**，自动连接 **Arty** 开发板，如图 **18-11** 和图 **18-12** 所示。

// 步骤三：右键 **FPGA Device**，选择“**Add Configuration Memory Device**”，如图 **18-13** 所示。

// 步骤四：选择如下参数的 **Flash**，如图 **18-14** 所示。

```
Part n25q128-3.3v
Manufacturer Micron
Family n25q
Type spi
Density 128
Width x1 x2 x4
```

// 步骤五：弹出“**Do you want to program the configuration memory device now?**”，选择 **OK**。

// 步骤六：在弹出的窗口中的<**Configuration file**>对话框中选择添加 <**your_e200_dir**>**/fpga/artydevkit/system.mcs**，然后选择“**OK**”，则开始烧写 **Flash**，可能会花费几十秒的时间等待。

// 步骤七：一旦烧写 **Flash** 成功，则可以通过按开发板上的“**PROG**”按键触发硬件电路使用 **Flash** 中的内容对 **FPGA** 重新进行烧录，烧录完成后“**PROG**”按键下方的 **DONE** 信号灯将变亮。

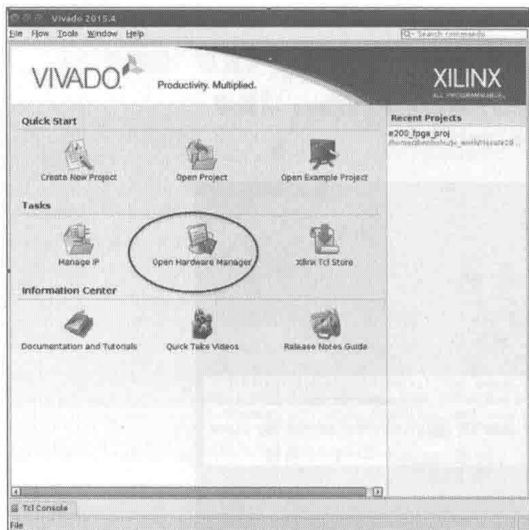


图 18-11　打开 Vivado Hardware Manager

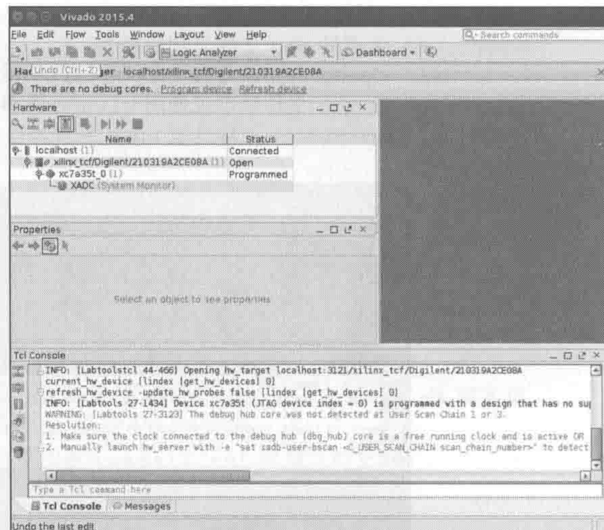


图 18-12　使用 Vivado Hardware Manager 连接 Arty 开发板

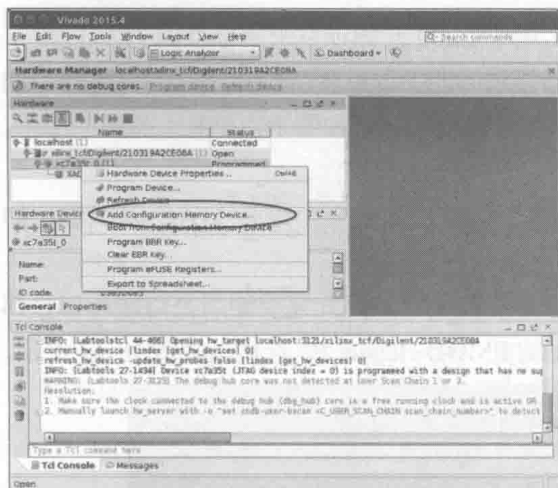


图 18-13 FPGA Device 选择 Add Configuration Memory Device

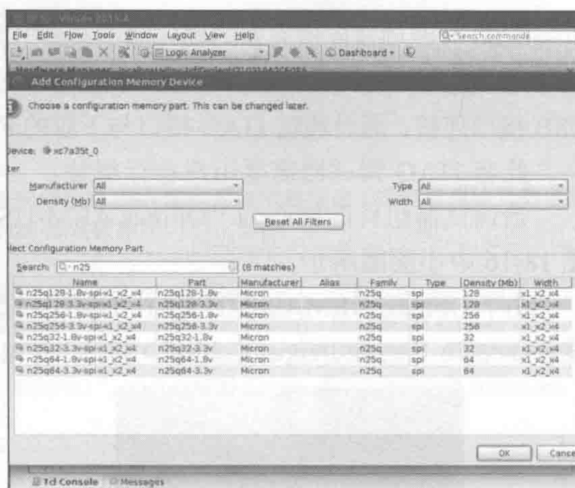


图 18-14 选择 Flash 类型

18.3.3 JTAG 调试器

为了支持使用 GDB 进行交互式调试或者通过 GDB 动态下载程序到处理器中运行，需要为 FPGA 原型平台配备一个 JTAG 调试器（JTAG Debugger）。在第 18.2 节中曾经介绍了 E200 处理器核支持通过标准的 JTAG 接口对其进行调试，在第 18.3.1 节中介绍了 SoC 顶层 JTAG 的 I/O Pad 连接到了 FPGA 芯片的 D4/D3 等 pin 脚上，而该组 pin 脚在 Arty 开发板上实际被连接到 PMOD Header JD 上，如图 18-15 中圆圈所示。

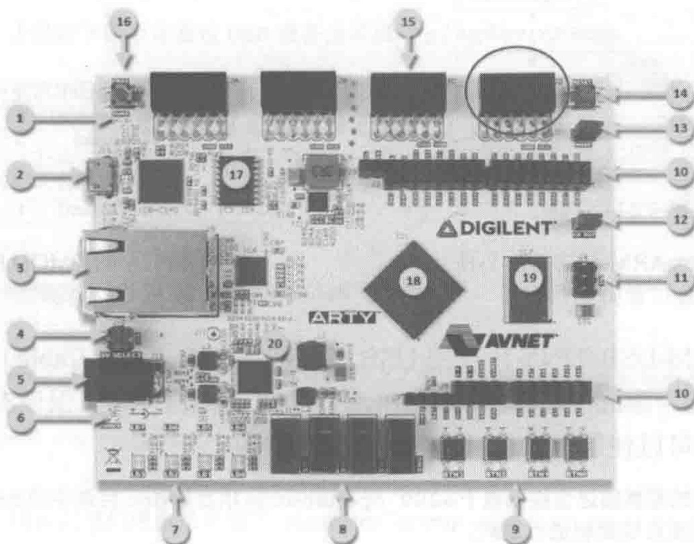


图 18-15 JTAG Pin 脚连接的 PMOD Header JD

E200 开源项目提供完整的硬件调试参考方案。由于 E200 开源项目采用标准的 JTAG 接口，因此需要使用 JTAG 调试器。该调试器通过 USB 转接线（USB A to B Cable）与上游主机 PC 的 USB 接口连接，同时通过 JTAG 接口与下游的 FPGA 开发板连接，如图 18-16 中大圆圈中所示。

此套 JTAG 调试器需要用户自行组装。

该调试器组件的核心为“Olimex ARM-USB-TINY-H”，这是一个硬件 JTAG 调试器，如图 18-16 中小圆圈所示。

由于 Olimex ARM-USB-TINY-H 的输出口是双排管脚接口，因此需要使用公口转母口杜邦线（Male-To-Female Jumper Cables），将其与 FPGA 的 PMOD Header JD 连接，如图 18-17 所示。

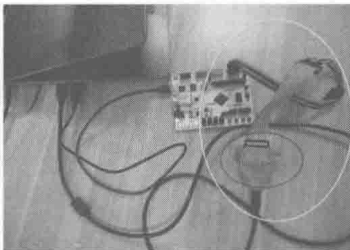


图 18-16 完整的 JTAG 调试器

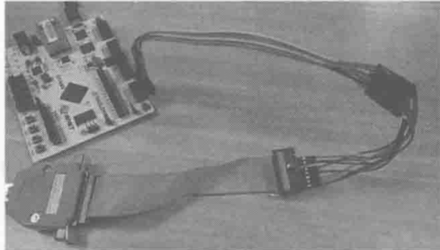


图 18-17 使用杜邦线连接 JTAG 调试器与 FPGA 开发板

为了方便连线且防止出错，推荐使用不同颜色的公口转母口连接线进行连接。图 18-18 所示是对 Olimex ARM-USB-TINY-H 输出的 20 根公口管脚进行了不同颜色的编号。图 18-19 所示是对 FPGA 的 PMOD Header JD 输入的 12 根母口管脚进行了不同颜色的编号。在连线时，严格使用对应颜色的连线逐一进行连接后（注意紧密连接防止接触不良），即宣告完成。

	1 : VREF (red)	2 : VREF (brown)
	3 : nTRST (orange)	4
	5 : TDI (yellow)	6
	7 : TMS (green)	8
NOTCH	9 : TCK (blue)	10
NOTCH	11	12
	13 : TDO (purple)	14 : GND (black)
	15 : nRST (grey)	16 : GND (white)
	17	18
	19	20
	LED	

图 18-18 Olimex ARM-USB-TINY-H 输出的 20 根公口管脚颜色编号

square pad	1 : TDO (purple)	7 : TDI (yellow)
	2 : nTRST (orange)	8 : TMS (green)
	3 : TCK (blue)	9 : nRST (grey)
	4	10
“GND”	5 : GND (black)	11 : GND (white)
“VCC”	6 : VREF (brown)	12 : VREF (red)

图 18-19 FPGA 的 PMOD Header JD 输入的 12 根母口管脚颜色编号

由于 Olimex ARM-USB-TINY-H 使用 USB 转接线（USB A to B Cable）将其与上游主机 PC 的 USB 接口连接，因此上游 PC 的 USB 端口需要正确的设置以保证有正确的权限。以 Ubuntu 16.04 为例，可以使用如下步骤进行配置。

```
// 注意：下列步骤的完整描述也被记载于e200_opensource项目的doc目录中的SoC_Quick_Start_
Guide文档，以便于读者直接复制进行重现。

// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置。
```

（1）使用 **VMware** 虚拟机在个人电脑上安装虚拟的 **Linux** 操作系统。
（2）由于 **Linux** 操作系统的版本众多，推荐使用 **Ubuntu 16.04** 版本的 **Linux** 操作系统有关如何安装 **VMware** 和 **Ubuntu** 操作系统本书不做介绍，有关 **Linux** 的基本使用本书也不做介绍，请读者自行查阅资料学习。

// 步骤二：将 **FPGA** 开发板通电（可以使用普通安卓手机 **USB** 充电器连接线供电，或者使用独立的电源供电）。使用 **USB A to B Cable** 将主机 **PC** 与 **FPGA** 开发板连接。

注意：

（1）务必使该 **USB** 接口被虚拟机的 **Linux** 系统识别（而非被 **Windows** 识别），如图 18-20 中圆圈所示，若 **USB** 图标在虚拟机中显示为高亮，则表明 **USB** 被虚拟机中 **Linux** 系统正确识别（而非被 **Windows** 识别）。

（2）若 **USB** 图标在虚拟机中显示为灰色，则表明 **USB** 没有被虚拟机中的 **Linux** 系统正确识别，如图 18-21 所示，可以使用鼠标点中 **USB** 图标，选择将其“连接（与主机的连接）”，将其连接至 **Linux** 系统（而非外部 **Windows**）。

// 步骤三：使用如下命令查看 **USB** 设备的状态。

```
lsusb      // 运行该命令后会显示如下信息。
 ...
Bus 001 Device 029: ID 15ba:002a Olimex Ltd. ARM-USB-TINY-H JTAG interface
```

// 步骤四：使用如下命令设置 **udev rules**，使得该 **USB** 设备能够被 **plugdev group** 所访问。

```
sudo vi /etc/udev/rules.d/99-openocd.rules
        // 用 vi 打开该文件，然后添加以下内容至该文件中，然后保存退出。
# These are for the Olimex Debugger for use with Arty Dev Kit
SUBSYSTEM=="usb", ATTR{idVendor}=="15ba",
ATTR{idProduct}=="002a", MODE="664", GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="15ba",
ATTRS{idProduct}=="002a", MODE="664", GROUP="plugdev"
```

// 步骤五：使用如下命令查看该 **USB** 设备是否属于 **plugdev group**。

```
ls /dev/ttyUSB*              // 运行该命令后会显示类似如下信息。
/dev/ttyUSB0 /dev/ttyUSB1

ls -l /dev/ttyUSB1          // 运行该命令后会显示类似如下信息。
crw-rw-r-- 1 root plugdev 188, 1 Nov 28 12:53 /dev/ttyUSB1
```

// 步骤六：将你自己的用户添加到 **plugdev group** 中。

```
whoami
            // 运行该命令能显示自己用户名，假设你的自己用户名显示为 your_user_name
            // 运行如下命令将 your_user_name 添加到 plugdev group 中
sudo usermod -a -G plugdev your_user_name
```

// 步骤七：确认自己的用户是否属于 **plugdev group**。

```
groups      // 运行该命令后会显示类似如下信息。
```

```
... plugdev ...
        // 只要从显示的 groups 中看到 plugdev 则意味着自己的用户属于该组，表示设置成功
```

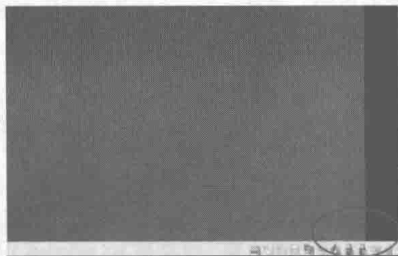


图 18-20 虚拟机 Linux 系统识别 USB 图标

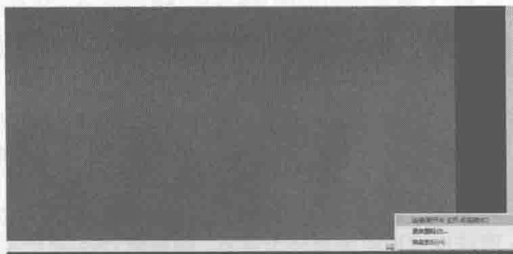


图 18-21 将 USB 接口选择连接至虚拟机中

JTAG 调试器的硬件组装工作至此便已完成。在第 19.4 节中将介绍利用该 JTAG 调试器如何使用 GDB 软件对 E200 开源 SoC 原型进行远程调试。

18.3.4 FPGA 原型平台 DIY 总结

将以上论述内容加以总结，为了能够搭建完整的 FPGA 原型平台，用户需要做如下硬件的准备工作。

- 购买一块 Xilinx Artix-7 35T Arty FPGA Evaluation Kit 开发板。
- 购买一根 USB A to Micro-B Cable（即安卓手机充电器 USB 线），用于给 FPGA 开发板供电且烧录 FPGA。
- 购买一块 Olimex ARM-USB-TINY-H Debugger。
- 购买一根 USB A to B Cable，用于连接主机 PC 与 Olimex ARM-USB-TINY-H Debugger。
- 购买一组 Male-To-Female Jumper Cables（杜邦线），用于连接 Olimex ARM-USB-TINY-H Debugger 与 FPGA 开发板。

用户需要做如下软件的准备工作。

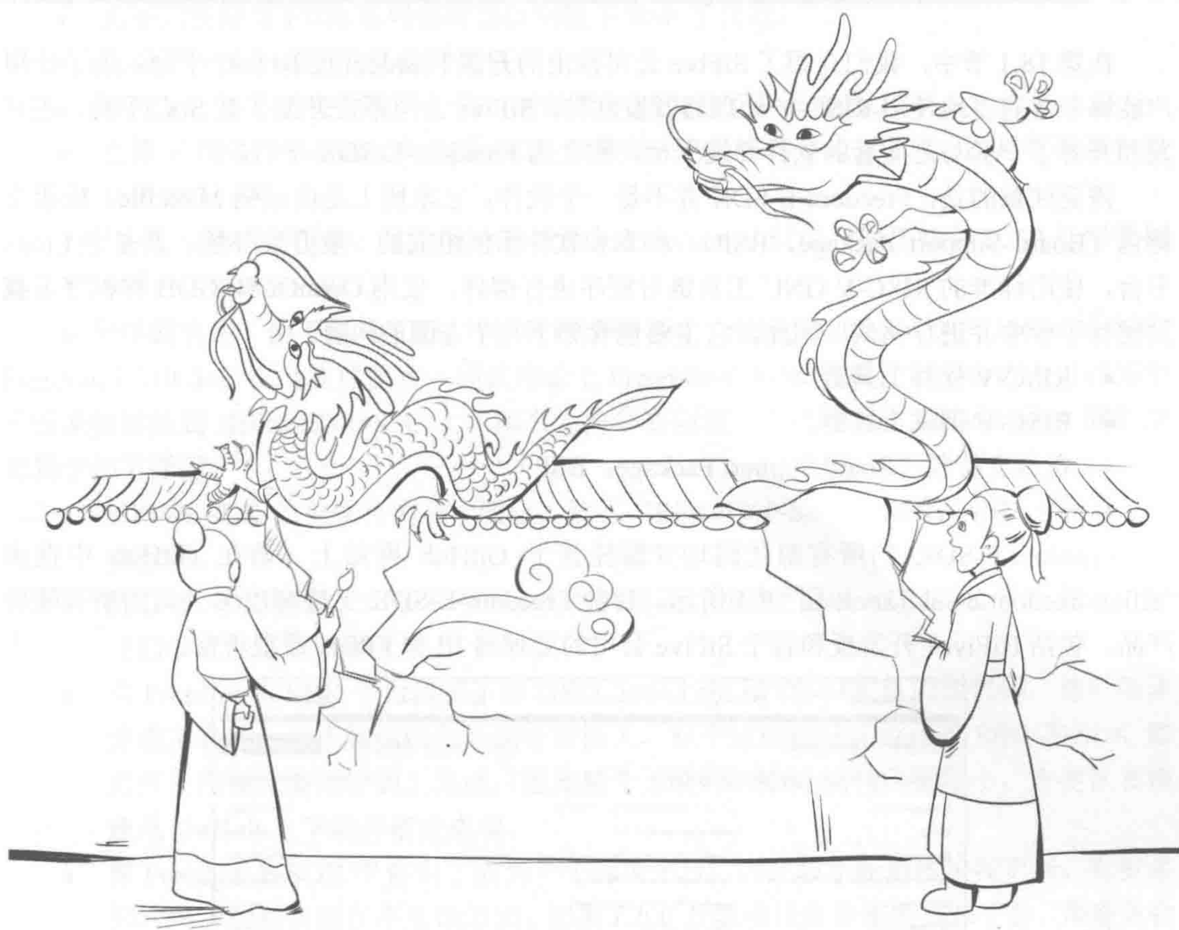
- 推荐安装 VMware 虚拟机，且安装 Linux 操作系统于虚拟机中。
- 安装 Xilinx 的 Vivado 软件，且安装 Arty 开发板的 board part。

在第 19 章中将介绍如何使用烧录后的 FPGA 原型平台运行真正的软件示例。

18.4 蜂鸟 E200 专用 FPGA 开发板

除了上一节中介绍的 Xilinx Arty FPGA 开发板，为了便于 RISC-V 爱好者学习和使用，蜂鸟 E200 会推出配套的入门级开发板，包括专门为蜂鸟 E200 定制的，具有极高性价比的专用 FPGA 开发板和 JTAG 调试器。感兴趣的读者可到 e200_opensource 项目的 boards 目录中了解蜂鸟 E200 专用开发板的相关信息。

第 19 章　画龙点睛——运行和调试软件示例



上一章中我们介绍了 E200 开源项目的配套 SoC 系统和代码，并且介绍了如何搭建 FPGA 原型平台。至此，“万事俱备，只欠东风”。

就像画龙点睛一样，只有当软件程序真正运行于处理器上面时，才意味着此处理器真的“活”了。如第 4 章所介绍的，蜂鸟 E200 提供完整的软件开发环境，更是难能可贵地提供交互式硬件调试工具（GDB）的支持。因此，本章将介绍如何使用 SoC 的 FPGA 原型平台运行真正软件示例，同时介绍如何使用 GDB 对程序进行调试。

19.1 Freedom-E-SDK 平台简介

在第 18.1 节中，我们介绍了 SiFive 公司推出的开源 Freedom E310 SoC 平台，为了让用户能够非常容易地使用 RISC-V 处理器开发软件，SiFive 公司不仅开源了其 SoC 平台，还开发和开源了一套与之配套的软件开发平台，称之为 Freedom-E-SDK 平台。

需要注意的是，Freedom-E-SDK 并不是一个软件，它本质上是由一些 Makefile、板级支持包（Board Support Package，BSP）、脚本和软件示例组成的一套开发环境。其基于 Linux 平台，使用标准的 RISC-V GNU 工具链对程序进行编译，使用 OpenOCD+GDB 将程序下载到硬件平台中并进行调试。因此，它主要包含如下几个方面的内容。

- RISC-V 软件工具链。
- RISC-V 调试工具链。
- 板级支持包（Board Support Package，BSP）。
- 若干软件示例。

Freedom-E-SDK 的所有源代码均开源托管于 GitHub 网站上（请在 GitHub 中搜索“sifive/freedom-e-sdk”），如图 19-1 所示。目前 Freedom-E-SDK 支持 SiFive 公司的所有硬件产品，包括 HiFive1 开发板和若干 SiFive 公司的处理器 IP 和 FPGA 原型平台。

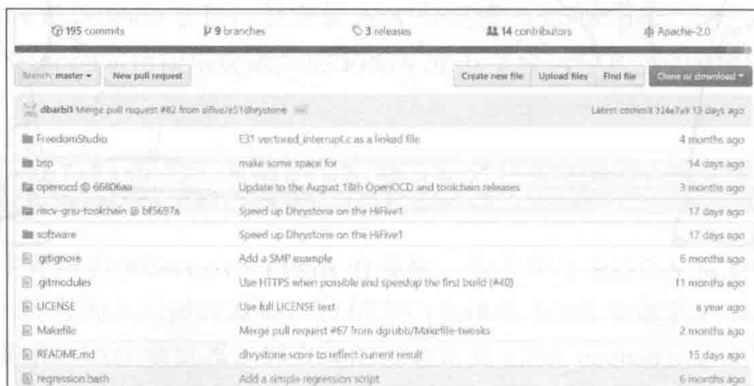


图 19-1　Freedom-E-SDK 的 GitHub 网站

19.2 SIRV-E-SDK 平台简介

19.2.1 SIRV-E-SDK 简介

Freedom-E-SDK 提供非常优秀的嵌入式软件开发环境，感谢 SiFive 公司将其开源。为了让用户能够轻松使用蜂鸟 E200 处理器核开发软件，E200 开源项目也以 Freedom-E-SDK 为蓝本，在其基础上做了如下主要修改。

- 为不同的蜂鸟 E200 系列处理器核创建了 BSP 子目录。
- 去除了一些不必要的目录和文件。
- 去除了一些不相关的软件示例，对软件示例进行了若干修改。
- 去除了 GNU Toolchain 和 OpenOCD 源代码，无须编译生成工具链，而是使用预先编译好的工具链。

为了方便读者理解区别，本书将“修改后的 Freedom-E-SDK（服务蜂鸟 E200 处理器核系列）”称为“SIRV-E-SDK”。

值得强调的是，由于蜂鸟 E200 开源处理器和 SoC（HBird-E200-SoC）与 SiFive 公司的 Freedom E310 SoC 完全软件兼容，因此理论上 Freedom-E-SDK 软件开发平台应该可以完全无缝地被移植到 HBird-E200-SoC 上，那么为何还要创建一个修改版的 SIRV-E-SDK 呢？主要基于如下原因。

- SIRV-E-SDK 的主要目的在于演示，演示 HBird-E200-SoC 的 FPGA 原型平台运行示例程序。
- SIRV-E-SDK 删除了一些不相干的文件和目录，相比 Freedom-E-SDK 更加简洁，方便初学者能够理解和上手。
- 原 Freedom-E-SDK 平台需要下载 GNU Toolchain 和 OpenOCD 的源代码，然后编译生成工具链，由于其源代码体积非常巨大，整个过程耗时耗力。而 SIRV-E-SDK 则直接使用预先编译好的工具链，因此整个 SIRV-E-SDK 的代码量很小，方便读者快速从 GitHub 上下载并搭建成型。
- 原 Freedom-E-SDK 平台为了成为一个通用平台，因此被不断地维护和更新，有更多的软件示例和功能在不断地添加。如果 E200 开源项目直接使用其源平台，则难免会出现某些更新过程中带来的错误。因此 SIRV-E-SDK 更追求稳定，一旦稳定后将停止修改，以方便用户能够稳定地使用 HBird-E200-SoC 进行示例软件的移植和演示，帮助用户更好地学习蜂鸟 E200 处理器核。

注意：SIRV-E-SDK 的初衷只是为了演示简单的软件示例并使其稳定，以方便用户在 GitHub 中下载使用，蜂鸟 E200 将专门维护一个独立的 GitHub 仓库（请在 GitHub 中搜索“hbird-e-sdk”），用于管理和维护蜂鸟 E200 的软件开发套件（SDK），并配套相关文档进行系统讲解。

19.2.2 SIRV-E-SDK 代码结构

SIRV-E-SDK 目录位于 e200_opensource 下的一个单独子目录，其代码结构如下所示。

```
e200_opensource
   |----sirv-e-sdk                // 存放 sirv-e-sdk 的目录
        |----bsp                  // 存放板级支持包（Board Support Package）的目录
             |----drivers         // 存放底层驱动代码的目录
             |----env             // 存放不同平台的配置文件夹
                  |----sirv-e201-arty    // 基于 E201 Core 平台的配置文件夹
                  |----sirv-e203-arty    // 基于 E203 Core 平台的配置文件夹
                  |----sirv-e205-arty    // 基于 E205 Core 平台的配置文件夹
                  |----sirv-e205fd-arty  // 基于 E205fd Core 平台的配置文件夹
             |----include              // 存放一些头文件
             |----libwrap              // 存放一些库文件
             |----tools                // 存放一些工具脚本文件
        |----software                  // 存放示例程序的源代码
             |----demo_gpio            // GPIO 示例程序
             |----coremark             // CoreMark 跑分程序
             |----dhrystone            // Dhrystone 跑分程序
        |----work                      // 存放工具链的目录
        |----Makefile                  // 主 Makefile 文件
```

各个主要的代码模块简述如下。

- bsp/drivers 目录主要用于一些驱动程序代码，譬如 PLIC 模块的底层驱动函数和代码。
- bsp/include 目录下主要存放包含 SoC 中外设模块的寄存器地址等参数的头文件。
- bsp/libwrap 目录主要存放一些与硬件平台相关的底层库函数的具体实现，这是嵌入式开发平台为了能够支持完整的 C/C++标准库函数而必须进行的移植工作。譬如最典型的 printf 函数，由于在嵌入式平台中没有显示屏，常见的做法是将嵌入式开发板通过 UART 接口与主机 PC 的串口连接，然后将 printf 函数打印的信息通过主机 PC 的串口打印显示在主机的电脑屏幕上。因此需要将 printf 库函数调用的最底层函数 write 函数进行移植，最底层的 write 函数将向 UART 的某些寄存器发起写操作，从而通过 UART 发送字符串至主机 PC 串口。bsp/libwrap 目录下存放的便是最底层函数的移植实现。
- bsp/env 目录主要用于存放不同 board 的相关支持包，譬如 bsp/env/sirv-e203-arty 文件夹存放的是使用 Arty FPGA 开发板（基于 E203 处理器核的 SoC）的支持包。另外，bsp/env 目录下还存放其他支持文件，譬如 common.mk 作为一个公用的 Makefile 脚本，start.S 作为程序的上电引导程序，和其他的若干头文件。

- software 目录主要存放软件示例，包括一个基本的 demo_gpio 示例、dhrystone 跑分程序和 CoreMark 跑分程序。每个示例均有其单独的文件夹，包含了各自的源代码、Makefile 和编译选项（在 Makefile 中指定）等。

19.3 使用 SIRV-E-SDK 运行示例程序

E200 开源项目提供一个典型的示例程序 demo_gpio，可运行于第 18.3 节中介绍的 Xilinx Arty 开发板上（基于 HBird-E200-SoC），可以使用 SIRV-E-SDK 平台按照如下步骤运行。

```
// 注意：下列步骤的完整描述也被记载于 e200_opensource 项目的 doc 目录中的 SoC_Quick_Start_
Guide 文档，以便于读者直接复制进行重现。

// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置。
        （1）使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。
        （2）由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统
        有关如何安装 VMware 以及 Ubuntu 操作系统本书不做介绍，有关 Linux 的基本使用本书
        也不做介绍，请读者自行查阅资料学习。

// 步骤二：将 e200_opensource 项目下载到本机 Linux 环境中，使用如下命令。

git clone https://github.com/SI-RISCV/e200_opensource.git
        // 经过此步骤将项目克隆下来，本机上即可具有如第 17.1 节中所述完整的
        // e200_opensource 目录文件夹，假设该目录为<your_e200_dir>，后文将使用该缩
        // 写指代。

// 步骤三：由于编译软件程序需要使用到 GNU 工具链，假设使用完整的 riscv-tools 来自己编译 GNU
工具链则费时费力，因此本书推荐使用预先已经编译好的 GCC 工具链。作者已经将工具链上传至网盘，网盘
具体地址记载于 e200_opensource 项目 prebuilt_tools 目录的 README 中，用户可以在网盘中的
“RISC-V Software Tools/RISC-V_GCC_201801_Linux”目录下载压缩包 gnu-mcu-eclipse-riscv-
none-gcc-7.2.0-2-20180111-2230-centos64.tgz 和 gnu-mcu-eclipse-openocd-0.10.0-6-
20180112-1448-centos64.tgz，然后按照如下步骤解压使用。（注意：上述链接网盘上的工具链可能会
不断更新，用户请注意自行判断使用最新日期的版本，下列步骤仅为特定版本的示例。）

cp gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz ~/
cp gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-centos64.tgz ~/
        //将两个压缩包均复制到用户的根目录下

cd ~/
tar -xzvf gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz
tar -xzvf gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-centos64.tgz
    // 进入根目录并解压上述两个压缩包，解压后可以看到一个生成的 gnu-mcu-eclipse 文件夹
```

```
cd <your_e200_dir>/sirv-e-sdk
      // 进入 e200_opensource 的 sirv-e-sdk 目录文件夹
mkdir -p work/build/openocd/prefix
      // 在 sirv-e-sdk 目录下创建上述这个 prefix 目录
cd work/build/openocd/prefix
      // 进入 prefix 该目录

ln -s ~/gnu-mcu-eclipse/openocd/0.10.0-6-20180112-1448/bin bin
      // 将用户根目录下解压的 OpenOCD 目录下的 bin 目录作为软链接链接到
      // 该 prefix 目录下

cd <your_e200_dir>/sirv-e-sdk
      // 再次进入 e200_opensource 的 sirv-e-sdk 目录文件夹
mkdir -p work/build/riscv-gnu-toolchain/riscv32-unknown-elf/prefix/
      // 在 sirv-e-sdk 目录下创建上述这个 prefix 目录
cd work/build/riscv-gnu-toolchain/riscv32-unknown-elf/prefix
      // 进入 prefix 该目录

ln -s ~/gnu-mcu-eclipse/riscv-none-gcc/7.2.0-2-20180111-2230/bin bin
      // 将用户根目录下解压的 GNU Toolchain 目录下的 bin 目录作为软链接链接到
      // 该 prefix 目录下
```

// 步骤四：按照第 18.3 节中所述方法，准备好基于 HBird-E200-SoC 的 Xilinx Arty FPGA 原型开发板，并将 bitstream 文件或者 mcs 文件烧录至 FPGA 中 FPGA 通电待命，且用 JTAG 调试器将 Arty 开发板与主机 PC 连接，并确保 JTAG 调试器的 USB 接口被虚拟机 Linux 系统正确识别。

// 步骤五：编译 demo_gpio 示例程序，使用如下命令。

```
cd <your_e200_dir>/sirv-e-sdk
      // 再次进入 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。

make software PROGRAM=demo_gpio BOARD=sirv-e203-arty
      // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包编译
      // demo_gpio 示例程序。
```

// 步骤六：将编译好的 demo_gpio 程序下载至 FPGA 原型开发板中，使用如下命令。

```
cd <your_e200_dir>/sirv-e-sdk
      // 再次进入 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。

make upload PROGRAM=demo_gpio BOARD=sirv-e203-arty
      // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包下载
      // demo_gpio 示例程序至 FPGA 开发板中。
      // 该过程的原理是调用 GDB 和 OpenOCD 软件通过 JTAG 调试器将编译好的 demo_gpio
      // 程序下载到 E203 处理器核中。
      // 程序若下载成功，则显示如图 19-2 所示。
```

```
// 步骤七：在 FPGA 原型平台上运行 demo_gpio 程序。

        // 由于 demo_gpio 示例程序将通过 UART 经过 FPGA 开发板的 mini-USB 接口
        // 连接至主机 PC，成为一个串口，打印一个 Log 符号到主机 PC 的显示屏上。
        // 因此需要先将串口显示终端准备好，在 Ubuntu 的命令行终端中使用如下命令。
sudo screen /dev/ttyUSB1 115200
        // 该命令将设备 ttyUSB1 设置为串口显示的来源，波特率为 115200
        // 若该命令执行成功的话，Ubuntu 的该命令行终端将被锁定，用于显示串口发送的字符。
        // 注意：
        // 若该命令无法执行成功，请检查如下几项。
        //   (1) 确保已按照第 18.3.3 节中所述方法将 USB 的权限设置正确。
        //   (2) 确保已按照第 18.3.3 节中所述方法将 USB 被 Linux 虚拟机识别（右下角
        //       显示为高亮）。
        //   (3) 按照 18.3.3 节中所述使用命令“ls /dev/ttyUSB*”查看 USB 被识别成为
        //       ttyUSB1 还是 ttyUSB2，若被识别成为 ttyUSB2，则应使用命令 sudo screen
        //       /dev/ttyUSB2 115200

        // 将主机 PC 的串口显示终端准备好之后，则仅需按 FPGA 原型开发板上的 RESET 按键
        // 即可。
```

按 FPGA 开发板上的 RESET 按键，则处理器复位开始执行 demo_gpio 程序，并将 Log 字符打印至主机 PC 的串口显示终端上，如图 19-3 所示。

由于本书侧重于介绍 CPU 的硬件设计，因此对于软件部分在此不做赘述，请读者自行阅读 demo_gpio 的代码了解其程序细节。

```
cleared protection for sectors 64 through 255 on flash bank 0
Info : JTAG tap: riscv.cpu tap/device found: 0x10e31913 (mfg: 0x489 (<unknown>), part
riscv.cpu: target state: halted
halted at 0x80003dca due to debug interrupt
Loading section .init, size 0xca lma 0x20000000
Loading section .text, size 0xbe78 lma 0x200000cc
Loading section .rodata, size 0xdec lma 0x2000bf44
Loading section .data, size 0x918 lma 0x2000cd30
Info : Padding image section 0 with 2 bytes
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
Info : JTAG tap: riscv.cpu tap/device found: 0x10e31913 (mfg: 0x489 (<unknown>), part
riscv.cpu: target state: halted
halted at 0x80000004 due to software breakpoint
Start address 0x20000000, load size 54854
Transfer rate: 20 KB/sec, 7836 bytes/write.
halted at 0x20000004 due to step
halted at 0x20000004 due to step
shutdown command invoked
shutdown command invoked
A debugging session is active.

        Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n) [answered Y; input not from terminal]
Remote communication error.  Target disconnected.: Connection reset by peer.
Successfully uploaded 'demo_gpio' to sirv-e203-arty.
```

图 19-2 下载 demo_gpio 程序至 FPGA 开发板成功后的显示界面

图 19-3 运行 demo_gpio 程序后在主机 PC 的串口显示终端上的 Log 字符

19.4 使用 GDB 和 OpenOCD 调试示例程序

GNU Project Debugger（GDB）是 GNU 工具链中的调试软件。GDB 是一款应用非常广泛的调试工具，能够用于调试 C、C++、Ada 等各种语言编写的程序，它提供如下功能。

- 下载或者启动程序。
- 通过设定各种特定条件来停止程序。
- 查看处理器的运行状态，包括通用寄存器的值、存储器地址的值等。
- 查看程序的状态，包括变量的值、函数的状态等。
- 改变处理器的运行状态，包括通用寄存器的值、存储器地址的值等。
- 改变程序的状态，包括变量的值、函数的状态等。

GDB 可以用于在主机 PC 的 Linux 系统中调试运行的程序，同时也能用于调试嵌入式硬件。在嵌入式硬件的环境中，由于资源有限，一般的嵌入式目标硬件上无法直接构建 GDB 的调试环境（譬如显示屏和 Linux 系统等），这时可以通过 GDB+GdbServer 的方式进行远程（Remote）调试，通常 GdbServer 在目标硬件上运行，而 GDB 则在主机 PC 上运行。

为了能够支持 GDB 对其进行调试，HBird-E200-SoC 使用 OpenOCD 作为其 GdbServer，与 GDB 进行配合。Open On-Chip Debugger（OpenOCD）是一款开源的免费调试软件，由社区共同维护。由于其开放开源的特点，众多的公司和个人使用其作为调试软件，支持大多数主流的 MCU 和硬件开发板。通过编写 OpenOCD 的底层驱动文件能够使其通过 JTAG 接口连接 HBird-E200-SoC，并利用其硬件调试特性对 HBird-E200-SoC 进行调试。请参阅第 14 章了解关于硬件调试的具体实现细节。

为了能够完全支持 GDB 的功能，在使用 GCC 对源代码进行编译时，需要使用-g 选项，例如：'gcc -g -o hello hello.c'。-g 选项会将调试所需信息加入编译所得的可执行程序中，因此该选项会增大可执行程序的大小，在正式发布的版本中通常不使用该选项。

GDB 虽然可以使用一些前端工具实现图形化界面，但是更常见的是使用命令行直接对其进行操作。常用的 GDB 命令介绍如表 19-1 所示。

表 19-1 GDB 常用命令

命令	介绍
load file	动态链入 file 文件，并读取它的符号表
jump	使当前执行的程序跳转到某一行，或者跳转到某个地址
info br	使用该指令可查看断点信息，br 是断点 break 的缩写，GDB 具有自动补齐功能，此命令等效于 info break
info source	使用该指令可查看当前程序的信息
info stack	使用该指令可查看程序的调用层次关系

续表

命　令	介　绍
list function-name	使用该指令可列出某个函数
list line-number	列出某行附近的代码
break function break line-number	在指定的函数，或者行号处设置断点
break *address	在指定的地址处设置断点，一般在没有源代码时使用
continue	恢复程序运行，直到遇到下一个断点
step	进入下一行代码的执行，会进入函数内部
step number	等效于连续执行 number 次 step 命令
next	执行下一行代码，但不会进入函数内部
next number	等效于连续执行 number 次 next 命令
until until line-number	继续运行直到到达指定行号，或者函数、地址等
stepi nexti	stepi/nexti 命令与 step/next 的区别在于其执行下一条汇编指令，而不是下一行代码（譬如 C/C++中的一行代码）
x address	打印指定存储器地址中的值
p variable	打印指定变量的值

按照如下步骤使用 GDB 和 OpenOCD 对基于 Xilinx Arty 开发板的 HBird-E200-SoC 原型平台进行调试。

```
    // 注意：下列步骤的完整描述也被记载于 e200_opensource 项目的 doc 目录中的 SoC_Quick_Start_
Guide 文档，以便于读者直接复制进行重现。

    // 步骤一 ～ 步骤四：与第 19.3 节中描述的运行 demo_gpio 示例程序步骤一 ～ 步骤四相同。

    // 步骤五：使用 GDB+OpenOCD 远程调试 FPGA 原型开发板，使用如下命令。

cd <your_e200_dir>/sirv-e-sdk
        // 再次进入到 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。

        // 使用如下命令打开 OpenOCD
make run_openocd PROGRAM=demo_gpio BOARD=sirv-e203-arty
        // 运行该命令会使用 bsp/env 目录下的 sirv-e203-arty 板级支持包中 OpenOCD 的
        // 配置文件来打开 OpenOCD，并与 Arty FPGA 开发板相连。
        // 如果该步骤执行成功，则如图 19-4 所示。

        // 由于命令行界面已经被 OpenOCD 挂住，因此需要重新开启一个新的 Terminal 终端，
        // 注意：再次强调，此处是重新开启一个新的 Terminal 终端。
        // 在新的 Terminal 终端下，确保进入到 e200_opensource 目录文件夹下面
        // 的 sirv-e-sdk 目录。
cd <your_e200_dir>/sirv-e-sdk
        // 然后使用如下命令打开 GDB
make run_gdb PROGRAM=demo_gpio BOARD=sirv-e203-arty
```

```
        // 运行该命令会自动打开 GDB 来调试 demo_gpio 示例程序。
        // 如果该步骤执行成功，则进入了 GDB 的调试命令行界面，如图 19-5 所示。
```

// 步骤六：演示使用 GDB 命令。

```
        // 接下来便可使用 GDB 的常用命令进行调试。
b main
        // 在 main 函数的入口处设置断点。

info b
        // 查看目前程序设置的断点，显示如图 19-6 所示。

x 0x20400000
x 0x20400004
x 0x20400008
        // 查看存储器地址 0x20400000/0x20400004/0x20400008 中的数值，显示如图 19-7
        // 所示。
info reg
info reg mstatus
        // 查看当前处理器的通用寄存器的值和 CSR 寄存器 mstatus 的值，显示如图 19-8 所示。

info reg csr768
        // 查看当前处理器的地址 768 的 CSR 寄存器的值。
        //  注意：编号 768 为十进制数，对应十六进制为 0x300，对应 mstatus 寄存器的 CSR
        //  地址。参见附录 B 了解 RSIC-V 架构的 CSR 寄存器列表和地址。

info reg mcause
info reg mepc
info reg mtval
        // 查看当前处理器的 CSR 寄存器 mcause，mepc 和 mtval 的值。
        //  注意：当程序出现了异常（程序运行结果显示结果为 Trap）时，可以通过 GDB 查看此
        //  3 个寄存器的值有效的定位异常的原因和发生位置。有关 mcause、mepc 和 mtval
        //  寄存器的详情，请参见附录 B2。

jump main
        // 从程序的 main 入口开始执行，将停于设置的第一个断点处，显示如图 19-9 所示。

ni
        // 单步执行，显示如图 19-10 所示。

continue
        // 继续执行，将停于下一个断点处，若无断点，则一直执行至程序结束处。
```

```
work/build/openocd/prefix/bin/openocd -f bsp/env/sirv-e203-arty/openocd.cfg
Open On-Chip Debugger 0.10.0-dev-g9bab078 (2017-02-02-01:39)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 1000 kHz
Info : auto-selecting first available session transport "jtag". To override use 'tran
Info : clock speed 1000 kHz
Info : JTAG tap: riscv.cpu tap/device found: 0x10e31913 (mfg: 0x489 (<unknown>), part
Info : Examined RISCV core; XLEN=32, misa=0x40001105
riscv.cpu: target state: halted
halted at 0x8000008e due to debug interrupt
Info : Found flash device 'micron n25q128' (ID 0x0018ba20)
cleared protection for sectors 0 through 255 on flash bank 0
```

图 19-4 打开 OpenOCD 后的命令行界面

```
GNU gdb (GDB) 7.12.50.20170109-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv32-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from software/demo_gpio/demo_gpio...done.
Remote debugging using localhost:3333
0x8000008e in main (argc=<optimized out>, argv=<optimized out>) at demo_gpio.c:292
292         static_tetes_xor_explicit(GPIO_REG(GPIO_OUTPUT_VAL), bitbang_mask, memo
(gdb)
```

图 19-5 GDB 的命令行界面

图 19-6　GDB 显示设置的断点

图 19-7　通过 GDB 查看存储器中的数据

图 19-8　GDB 显示寄存器的值

图 19-9　GDB 显示程序停止于断点处

图 19-10　GDB 单步执行程序

19.5 Windows 图形化 IDE 开发工具

除了本章介绍的 SDK 软件平台（在 Linux 环境中使用命令行和脚本操作）之外，目前 RISC-V 还提供成熟的、基于 Eclipse Windows 图形化开发界面的集成开发环境（Integrated Development Environment，IDE）。

由于 Linux 操作系统在很多工程领域更受推崇，其使用的命令行和脚本操作使得项目具备更高的可重现性和高效的自动化特性，因此作者更推荐基于 Linux 环境的 SDK 软件开发平台。有关基于 Windows IDE 工具内容，本书在此不做赘述，感兴趣的读者可以参见 e200_opensource 项目 doc 目录中有关 Windows IDE 工具的使用说明。

第 20 章　是骡子是马？拉出来遛遛
——运行跑分程序

猜猜谁是马



本书至此已经通过蜂鸟 E200 处理器核为实例详细介绍了 CPU 的硬件设计技巧，同时也介绍了如何仿真，以及如何在 FPGA 原型平台上运行真实的软件示例。想必读者已经从硬件到软件上均有了充分的认识。

本章将对处理器的效能加以评估，衡量处理器的一个重要指标是功耗，另一个重要指标便是性能。

对于功耗的评估，其大致与处理器的硬件面积呈正比，本书在第二部分中的各章节详细论述了蜂鸟 E200 处理器的微架构和设计细节，在第 4.4 节列举了蜂鸟 E200 处理器的面积数据，可以看出蜂鸟 E200 是一款极为精简的小面积超低功耗处理器，本书在此不再赘述。

那么如何衡量处理器设计的性能呢？所谓“是骡子是马？拉出来遛遛”就知道了，那么对于处理器性能的评估，跑一跑跑分程序就知道了。

20.1 跑分程序简介

跑分程序（Benchmarks）通常是一组标准的软件程序，让处理器运行该标准程序，并通过运行速度计算出一组分数，作为衡量处理器性能的指标。

跑分程序通常由标准的高级语言（譬如 C/C++）编写，与底层硬件平台特性和指令集架构无关。因此各个不同架构或者不同厂商的处理器均可以运行相同的跑分程序，并可以根据其运行所得的分数来对其性能进行比较和衡量。

在处理器领域的跑分程序非常众多，有某些个人开发的，也有某些标准组织或者商业公司开发的跑分程序，本书在此不一一列举。

在嵌入式处理器领域最为知名和常见的跑分程序为 Dhrystone 和 CoreMark。本章将通过运行此两个跑分程序来衡量蜂鸟 E200 处理器的性能。

20.2 Dhrystone 简介

Dhrystone 是一个综合的处理器跑分程序，由 Reinhold P. Weicker 于 1984 年开发，用于衡量处理器的整数运算处理性能。在 Dhrystone 的程序中，作者收集了众多不同类型程序中的典型特性，采用了各种典型的方法，譬如函数调用、间接指针、赋值等，使得该程序测试的性能非常具有代表性。

最初版本的 Dhrystone 由 Ada 语言编写，之后的 C 语言对应版本由 Rick Richardson 开发，使得 Dhrystone 更加流行。由于被广泛采用，Dhrystone 成为了当今最有代表性的通用处理器跑分程序，几乎每一款 CPU 都必须公布其 Dhrystone 的跑分作为衡量其性能指标的重要参数。

熟悉计算机体系结构的读者应该了解性能指标 MIPS（Million Instructions Per Second）的含义，其反映了处理器在汇编指令级别执行的速度。由于高级语言（譬如 C/C++）编写的程序通过不同处理器架构的编译器编译后，生成的汇编代码可能会有巨大差别。譬如有的指令集架构的代码密度很高，能够产生少量的汇编指令便可完成程序，而有的指令集架构的代码密度比较低，则需要产生大量的汇编指令完成程序。因此单纯的 MIPS 指标仅能反映处理器执行汇编指令的硬件效率，而不能反映出处理器软硬件系统的综合性能。

而 Dhrystone 的跑分结果使用更加有意义的 Dhrystone Per Second 作为衡量标准，表示处理器每秒钟能够执行的 Dhrystone 主循环的次数。Dhrystone 程序的主循环由一个 For 循环组成，如图 20-1 所示，且可以通过参数控制具体的循环次数。For 循环内部则调用各种编写的子函数，这些子函数便是 Dhrystone 开发者刻意构造的具有代表性的程序代码，如图 20-2 所示。在 For 主循环的开始和结束部分均通过计时器（Timer）读取出当前的时间值，如图 20-3 所示，最后通过开始和结束时间值的差值得出运行特定循环次数的总执行时间。该总执行时间取决于如下两个方面的效率。

图 20-1　Dhrystone 程序片段一

图 20-2　Dhrystone 程序片段二

图 20-3　Dhrystone 程序片段三

- 一方面取决于指令集架构的效率和编译器的优劣，其决定了高层语言编写的 Dhrystone 程序能够编译成多少汇编指令。
- 另一方面取决于处理器的硬件性能，其决定了处理器能以多快的速度执行完这些指令。

综上可见，DMIPS 能够反映处理器从架构、编译器到硬件的综合性能。

Dhrystone 跑分结果另一种更常用的表示单位是 DMIPS（Dhrystone MIPS），其使用早期的 VAX 11/780 处理器作为标称值，定义如下。

- 由于 VAX 11/780 处理器被公认能达到 1 MIPS 的性能，使用其运行 Dhrystone 跑分程序能够达到的性能为 1757 Dhrystone Per Second。因此将其作为黄金参考，将 Dhrystone Per Second 除以 1757 所得值称为 1 DMIPS。
  譬如：假设某处理器能够每秒钟执行 2000000 次 Dhrystone 主循环，则其性能约等于 2000000/1757=1138 DMIPS。
- 在此基础之上，去除处理器主频的因素，假设处理器以 1MHz 的主频运行 Dhrystone 所得的 DMIPS 结果则为 DMIPS/MHz，该种表示方式也极为常见。
  譬如：假设前述处理器（1138 DMIPS）运行主频为 1GHz，则其性能指标也可表示为 1138/1000=1.138 DMIPS/MHz。

20.3 运行 Dhrystone Benchmark

Dhrystone Benchmark 可运行于第 18.2 节中介绍的 HBird-E200-SoC FPGA 原型平台中（基于 Xilinx Arty 开发板），使用第 19.2 节中介绍的 SIRV-E-SDK 软件平台按照如下步骤运行。

```
  // 注意：下列步骤的完整描述也被记载于 e200_opensource 项目的 doc 目录中的
SoC_Quick_Start_Guide 文档，以便于读者直接复制进行重现。

  // 步骤一 ~ 步骤四：与第 19.3 节中描述的运行 demo_gpio 示例程序步骤一 ~ 步骤四相同。


  // 步骤五：编译 Dhrystone 示例程序，使用如下命令。

  cd <your_e200_dir>/sirv-e-sdk
        // 确保进入 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。

  make software PROGRAM=dhrystone BOARD=sirv-e203-arty
        // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包编译
        // dhrystone 示例程序。

  // 步骤六：将编译好的 Dhrystone 程序下载至 FPGA 原型开发板中，使用如下命令。

  cd <your_e200_dir>/sirv-e-sdk
        // 确保进入 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。

  make upload PROGRAM=dhrystone BOARD=sirv-e203-arty
        // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包下载
        // dhrystone 示例程序至 FPGA 开发板中。
        // 该过程的原理是调用 GDB 和 OpenOCD 软件，通过 JTAG 调试器将编译好的 dhrystone
        // 程序下载到 E203 处理器核中。
```

```
// 步骤七：在 FPGA 原型平台上运行 Dhrystone 程序。

      // 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考第 19.3 节中
      // 运行 demo_gpio 示例程序的步骤七将串口显示电脑屏幕设置好，并使得程序的打印信
      // 息能够显示在电脑屏幕上。
      //
      // 由于步骤六已经将程序烧写进 FPGA 开发板的 Flash 之中，因此每次按 FPGA 开发板的
      // RESET 按键，则处理器复位开始执行 Dhrystone 程序，并将 Log 字符打印至主机 PC
      // 的串口显示终端上。从其打印的结果我们可以看出 E203 处理器运行 Dhrystone 程
      // 序的结果性能指标，如图 20-4 所示。
```

注意：目前蜂鸟 E200 系列中开源的处理器型号为 E203。因此读者无法运行其他型号的处理器核得到其 Dhrystone 分数。但本书将 E201 与 E205 的运行分数也在此加以列举，以方便对比。

图 20-4、图 20-5 和图 20-6 分别显示了 E203、E201 和 E205 运行 Dhrystone 所得跑分，从中可以看出 E205 得分 1.355 DMIPS/Hz 高于 E201 的分数 1.171 DMIPS/Hz。这是因为 E205 中使用了单周期的硬件乘法器和多周期的硬件除法器，而 E201 并没有硬件的乘法器和除法器。可见硬件乘除法单元对于 Dhrystone 跑分有一定的帮助。

```
Ch_2_Glob:              B
        should be:      B
Arr_1_Glob[8]:          7
        should be:      7
Arr_2_Glob[8][7]:       160010
        should be:      Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:             -1879031808
        should be:      (implementation-dependent)
  Discr:                0
        should be:      0
  Enum_Comp:            2
        should be:      2
  Int_Comp:             17
        should be:      17
  Str_Comp:             DHRYSTONE PROGRAM, SOME STRING
        should be:      DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:             -1879031808
        should be:      (implementation-dependent), same as above
  Discr:                0
        should be:      0
  Enum_Comp:            1
        should be:      1
  Int_Comp:             18
        should be:      18
  Str_Comp:             DHRYSTONE PROGRAM, SOME STRING
        should be:      DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:              5
        should be:      5
Int_2_Loc:              13
        should be:      13
Int_3_Loc:              7
        should be:      7
Enum_Loc:               1
        should be:      1
Str_1_Loc:              DHRYSTONE PROGRAM, 1'ST STRING
        should be:      DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:              DHRYSTONE PROGRAM, 2'ND STRING
        should be:      DHRYSTONE PROGRAM, 2'ND STRING

 (*) User_Cycle for total run through Dhrystone with loops 160000:
73920053
       So the DMIPS/MHz can be caculated by:
       1000000/(User_Cycle/Number_Of_Runs)/1757 = 1.231930 DMIPS/MHz


Progam has exited with code:0x00000000
```

图 20-4　E203 Core 运行 Dhrystone Benchmark 后于主机串口终端上显示分数

```
Bool_Glob:              1
        should be:      1
Ch_1_Glob:              A
        should be:      A
Ch_2_Glob:              B
        should be:      B
Arr_1_Glob[8]:          7
        should be:      7
Arr_2_Glob[8][7]:       40010
        should be:      Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:             -1879036032
        should be:      (implementation-dependent)
  Discr:                0
        should be:      0
  Enum_Comp:            2
        should be:      2
  Int_Comp:             17
        should be:      17
  Str_Comp:             DHRYSTONE PROGRAM, SOME STRING
        should be:      DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:             -1879036032
        should be:      (implementation-dependent), same as above
  Discr:                0
        should be:      0
  Enum_Comp:            1
        should be:      1
  Int_Comp:             18
        should be:      18
  Str_Comp:             DHRYSTONE PROGRAM, SOME STRING
        should be:      DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:              5
        should be:      5
Int_2_Loc:              13
        should be:      13
Int_3_Loc:              7
        should be:      7
Enum_Loc:               1
        should be:      1
Str_1_Loc:              DHRYSTONE PROGRAM, 1'ST STRING
        should be:      DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:              DHRYSTONE PROGRAM, 2'ND STRING
        should be:      DHRYSTONE PROGRAM, 2'ND STRING

 (*) User_Cycle for total run through Dhrystone with loops 40000:
19440052
       So the DMIPS/MHz can be caculated by:
       1000000/(User_Cycle/Number_Of_Runs)/1757 = 1.171091 DMIPS/MHz


Progam has exited with code:0x00000000
```

图 20-5　E201 Core 运行 Dhrystone Benchmark 后于主机串口终端上显示分数

```
Ch_1_Glob:          A
        should be:  A
Ch_2_Glob:          B
        should be:  B
Arr_1_Glob[8]:      7
        should be:  7
Arr_2_Glob[8][7]:   100010
        should be:  Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:         -1879036832
        should be:  (implementation-dependent)
  Discr:            0
        should be:  0
  Enum_Comp:        2
        should be:  2
  Int_Comp:         17
        should be:  17
  Str_Comp:         DHRYSTONE PROGRAM, SOME STRING
        should be:  DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:         -1879036832
        should be:  (implementation-dependent), same as above
  Discr:            0
        should be:  0
  Enum_Comp:        1
        should be:  1
  Int_Comp:         18
        should be:  18
  Str_Comp:         DHRYSTONE PROGRAM, SOME STRING
        should be:  DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:          5
        should be:  5
Int_2_Loc:          13
        should be:  13
Int_3_Loc:          7
        should be:  7
Enum_Loc:           1
        should be:  1
Str_1_Loc:          DHRYSTONE PROGRAM, 1'ST STRING
        should be:  DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:          DHRYSTONE PROGRAM, 2'ND STRING
        should be:  DHRYSTONE PROGRAM, 2'ND STRING

 (*) User_Cycle for total run through Dhrystone with loops 100000:
42000040
       So the DMIPS/MHz can be caculated by:
       1000000/(User_Cycle/Number_Of_Runs)/1757 = 1.355122 DMIPS/MHz


Progam has exited with code:0x00000000
```

图 20-6　E205 Core 运行 Dhrystone Benchmark 后于主机串口终端上显示分数

20.4 CoreMark 简介

CoreMark 也是一个综合的处理器 Benchmark 程序，由非营利组织 Embedded Microprocessor Benchmark Consortium（EEMBC）的 Shay Gal-On 于 2009 年开发。CoreMark 程序的源代码的体量与 Dhrystone 一样非常小，因此可以运行在包括极低功耗微处理器在内的各种处理器上。并且 EEMBC 网站免费提供 CoreMark 程序的源代码下载，目的是使 CoreMark 能够成为一种行业标准的 Benchmark 程序，以替代年代久远的 Dhrystone。

CoreMark 程序由 C 语言编写，包含了很多典型的算法，譬如链表操作、矩阵运算、状态机（用来确定输入流中是否包含有效数字）和循环冗余校验（CRC）。这些算法尤其在嵌入式领域的软件中极为常见，因此 CoreMark 在嵌入式领域被认为比 Dhrystone 更加具有代表意义，很多嵌入式领域的 CPU 都公布了其 CoreMark 的跑分作为衡量其性能指标的重要参数。

CoreMark 结果的表示方法与 Dhrystone 相似，使用 Number of iterations per second 作为衡量标准，表示处理器每秒钟能够执行的 CoreMark 主循环的次数。CoreMark 程序的主循环也由一个迭代循环组成，如图 20-7 所示，且可通过参数控制具体的循环次数。循环内部则调用各种编写的子函数，如图 20-8 所示。在主循环的开始和结束部分均通过计时器读取出

当前的时间值，如图 20-9 所示。最后通过开始和结束时间值的差值得出运行特定循环次数的总执行时间，并依此计算出单位时间内能够运行的循环次数。在此基础之上，除以处理器主频的因素，则可以计算出 CoreMark/Hz。

譬如：假设某处理器以 20MHz 的主频运行 CoreMark，程序能够达到每秒执行 50 次主循环则其性能为 50/20=2.5 CoreMark/MHz。

图 20-7　CoreMark 程序片段一

图 20-8　CoreMark 程序片段二

图 20-9　CoreMark 程序片段三

20.5 运行 CoreMark Benchmark

CoreMark Benchmark 可运行于第 18.2 节中介绍的 HBird-E200-SoC FPGA 原型平台（基于 Xilinx Arty 开发板）中，使用第 19.2 节中介绍的 SIRV-E-SDK 软件平台按照如下步骤运行。

```
// 注意：下列步骤的完整描述也被记载于 e200_opensource 项目的 doc 目录中的 SoC_Quick_Start_
Guide 文档，以便于读者直接复制进行重现。

// 步骤一 ~ 步骤四：与第 19.3 节中描述的运行 demo_gpio 示例程序步骤一 ~ 步骤四相同。

// 步骤五：下载 CoreMark 的源代码
        // 由于未经 EEMBC 运行不得擅自转发 CoreMark 程序的源代码，因此需要用户自行于
        // EEMBC 网站上下载 CoreMark 的程序源代码。
        // 将下载压缩包中的如下文件复制至 sirv-e-sdk/software/coremark 目录下。
                core_list_join.c
                core_main.c
                coremark.h
```

```
                core_matrix.c
                core_state.c
                core_util.c

// 步骤六：编译 CoreMark 示例程序，使用如下命令。

cd <your_e200_dir>/sirv-e-sdk
        // 再次进入 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。

make software PROGRAM=coremark BOARD=sirv-e203-arty
        // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包编译
        // coremark 示例程序。

// 步骤七：将编译好的 CoreMark 程序下载至 FPGA 原型开发板中，使用如下命令。

cd <your_e200_dir>/sirv-e-sdk
        // 再次进入 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。

make upload PROGRAM=coremark BOARD=sirv-e203-arty
        // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包下载
        // coremark 示例程序至 FPGA 开发板中。
        // 该过程的原理是调用 GDB 和 OpenOCD 软件，通过 JTAG 调试器将编译好的 coremark
        // 程序下载到 E203 处理器核中。

// 步骤八：在 FPGA 原型平台上运行 CoreMark 程序。

        // 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考第 19.3 节中
        // 运行 demo_gpio 示例程序的步骤七将串口显示电脑屏幕设置好，并使得程序的打印信
        // 息能够显示在电脑屏幕上。
        //
        // 由于步骤六已经将程序烧写进 FPGA 开发板的 Flash 之中，因此每次按 FPGA 开发板的
        // RESET 按键，则处理器复位开始执行 CoreMark 程序，并将 Log 字符打印至主机 PC
        // 的串口显示终端上。从其打印的结果我们可以看出 E203 处理器运行 CoreMark 跑分程
        // 序的结果性能指标，如图 20-10 所示。
```

注意：目前蜂鸟 E200 系列中开源的处理器型号为 E203。因此读者无法运行其他型号的处理器核得到其 CoreMark 分数。但本书将 E201 与 E205 的运行分数也在此加以列举，以方便对比。

图 20-10 E203 Core 运行 CoreMark Benchmark 后于主机串口终端上显示分数

图 20-10、图 20-11 和图 20-12 分别显示了 E203、E201 和 E205 运行 CoreMark 所得跑分，从中可以看出 E205 得分 3.327 CoreMark/Hz 远远高于 E201 的分数 1.352 CoreMark/Hz。这是因为 E205 中使用了单周期的硬件乘法器和多周期的硬件除法器，而 E201

并没有硬件的乘法器和除法器，可见硬件乘除法单元对于 CoreMark 的跑分帮助非常显著。

```
2K performance run parameters for coremark.
CoreMark Size     : 666
Total ticks       : 295853718
Total time (secs): 35
Iterations/Sec    : 11
Iterations        : 400
Compiler version : GCC6.1.0
Compiler flags    : -O2 -fno-common -funroll-loops -finline-functions --param max-inline-insn
s-auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4
Memory location   : STACK
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x25b5
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 11 / GCC6.1.0 -O2 -fno-common -funroll-loops -finline-functions --param max-i
nline-insns-auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4 / STACK


Print Personal Added Addtional Info to Easy Visual Analysis

 (*) Assume SIRV (Silicion Integrated Developped RISC V Core) running at 1 MHz
     So the CoreMark/MHz can be caculated by:
     (Iterations*1000000/total ticks) = 1.352019 CoreMark/MHz


Progam has exited with code:0x00000000
```

图 20-11　E201 Core 运行 CoreMark Benchmark 后于主机串口终端上显示分数

```
2K performance run parameters for coremark.
CoreMark Size     : 666
Total ticks       : 120224918
Total time (secs): 14
Iterations/Sec    : 27
Iterations        : 400
Compiler version : GCC6.1.0
Compiler flags    : -O2 -fno-common -funroll-loops -finline-functions --param max-inline-insn
s-auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4
Memory location   : STACK
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x25b5
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 27 / GCC6.1.0 -O2 -fno-common -funroll-loops -finline-functions --param max-i
nline-insns-auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4 / STACK


Print Personal Added Addtional Info to Easy Visual Analysis

 (*) Assume SIRV (Silicion Integrated Developped RISC V Core) running at 1 MHz
     So the CoreMark/MHz can be caculated by:
     (Iterations*1000000/total ticks) = 3.327097 CoreMark/MHz


Progam has exited with code:0x00000000
```

图 20-12　E205 Core 运行 CoreMark Benchmark 后于主机串口终端上显示分数

20.6 总结与比较

由于 Dhrystone 和 CoreMark 均只使用了整数运算类型，因此并不能衡量浮点运算处理性能。对于处理器的浮点性能指标，有众多专门的浮点 Benchmark 程序能够运行，本书在此不做介绍。

通过上述运行结果，可以总结蜂鸟 E200 处理器核的 Benchmark 分数与主流的 ARM Cortex-M 系列处理器的 Benchmark 分数对比，如表 20-1 所示。

从表 20-1 可以看出，蜂鸟 E203 功耗面积和性能均不逊色于 ARM 的 Cortex-M0+处理器核（M0+ 是 ARM 最小面积的处理器核心），蜂鸟 E205 功耗面积和性能均不逊色于 ARM 的 Cortex-M3 处理器核。

注意：有关 ARM Cortex-M 系列处理器核的性能数据来自于本书撰写之时收集的公开信息，非官方数据，本书对其正确性不做保证，请读者以最新 ARM 官方数据为准。

表 20-1　蜂鸟 E200 系列处理器核与 ARM Cortex-M 处理器核的 Benchmark 结果对比

Benchmarks	ARM Cortex-M0	ARM Cortex-M0+	ARM Cortex-M3	蜂鸟 E201	蜂鸟 E203	蜂鸟 E205
Dhrystone (DMIPS/MHz)	0.84 (Official) 1.21 (Max options)	0.94 (Official) 1.31 (Max options)	1.25	1.17	1.23	1.35
CoreMark (CoreMark/MHz)	2.33	2.42	3.32	1.35	2.14	3.32
最小配置逻辑门数（K Gates）	12K	12K	36K	10K	12K	20K

注：Cortex-M0+的乘法器可以配置成单周期乘法器或多周期迭代乘法器。Dhrystone 性能数据与 CoreMark 性能数据是采用单周期乘法还是多周期乘法器的信息不详

附录部分

RISC-V 架构详述

附录 A　RISC-V 架构指令集介绍

附录翻译自 RISC-V 的“指令集文档”，本书对相关内容进行了重组，以求通俗易懂。

A1　RV32GC 架构概述

当前 RISC-V 架构文档主要分为：

- “指令集文档”（riscv-spec-v2.2.pdf）
- “特权架构文档”（riscv-privileged-v1.10.pdf）

注意：以上文档版本号为本书撰写之时的最新版本，RISC-V 的架构文档还在不断地丰富和更新，但是指令集架构的基本面（本书介绍部分）已经确定，不会再修改。如第 2.1.1 节所述，读者可以在 RISC-V 基金会的网站上注册和关注，并免费下载其完整原文。

请参见第 2 章了解有关 RISC-V 指令集的特点和概述。RISC-V 指令集本身是模块化的指令集，可以灵活地进行组合，具有相当多的可配置型。蜂鸟 E200 处理器核系列支持如下模块化指令集。

- 32 位：32 位地址空间，通用寄存器宽度 32 位。
- I：支持 32 个通用整数寄存器。
- M：支持整数乘法与除法指令。
- A：支持存储器原子（Atomic）操作指令和 Load-Reserved/Store-Conditional 指令。
- F：支持单精度浮点指令。
- D：支持双精度浮点指令。
- C：支持编码长度为 16 位的压缩指令，提高代码密度。
- Machine Mode Only：只支持机器模式。

按照 RISC-V 架构命名规则，以上指令子集的组合可表示为 RV32IMAFDC。RISC-V 架构定义 IMAFD 为通用组合（General Purpose），以字母 G 表示，因此 RV32IMAFDC 也可表示为 RV32GC。

RV32GC 是最常见的 32 位 RISC-V 指令集组合，因此附录仅介绍 RV32GC 相关的指令集，以便读者快速学习并掌握 RISC-V 架构的基本指令集知识。关于本书未予介绍的其他指令子集细节，感兴趣的读者请参见 RISC-V 架构的“指令集文档”原文。

A2 RV32E 架构概述

RISC-V 提供一种可选的嵌入式架构（由字母 E 表示），仅需 16 个通用整数寄存器组成寄存器组，主要用于追求极低面积与极低功耗的嵌入式场景。

除此之外，RISC-V 架构文档中对嵌入式架构提供了一些其他的约束和建议。

- 嵌入式架构仅支持 32 位架构，在 64 或 128 位架构中不支持该嵌入式架构。即只有 RV32E，而没有 RV64E。
- 在嵌入式架构中推荐使用压缩指令子集（由字母 C 表示），即 RV32EC，以增加嵌入式系统中关注的代码密度。
- 在嵌入式架构中不支持浮点指令子集。如果需要选择支持浮点指令子集（F 或者 D），则必须使用非嵌入式架构（RV32I 而非 RV32E）。
- 嵌入式架构仅支持机器模式（Machine Mode）与用户模式（User Mode），不支持其他的特权模式。
- 嵌入式架构仅支持直接的物理地址管理，而不支持虚拟地址。

除了上述约束之外，RV32E 的其他特性与基本的整数指令架构（RV32I）完全相同，因此本书对 RV32E 架构不再赘述。

A3 蜂鸟 E200 支持的指令列表

注意：由于蜂鸟 E200 是一个处理器系列，并非每一个型号的蜂鸟 E200 处理器核均支持 A1 中所述的所有指令子集。以开源的 E203 处理器核为例，由于它默认支持的架构为 RV32IMAC，因此其仅支持 RV32IMAC 相关的指令子集，其他型号（E201/E205/E205f/E205fd）同理。有关蜂鸟 E200 处理器的不同型号及其默认支持的架构，请参见第 4.3 节。

A4 寄存器组

在 RISC-V 架构中，寄存器组主要包括通用寄存器（General Purpose Registers）和控制状态寄存器（Control and Status Register，CSR）。

A4.1 通用寄存器组

对于通用寄存器组，RISC-V 架构规定如下。

- 如果使用的是基本整数指令子集（由字母 I 表示），那么 RISC-V 架构包含 32 个通用整数寄存器，由代号 x0～x31 表示。

其中通用整数寄存器 x0 被预留为常数 0，其他 31 个（x1～x31）为普通的通用整数寄存器。

在 RISC-V 的架构中，通用寄存器的宽度由 XLEN 这个术语表示。如果是 32 位架构（由 RV32I 表示），每个寄存器的宽度为 32 位；如果是 64 位架构（由 RV64I 表示），每个寄存器的宽度为 64 位。

- 如果使用的是嵌入式架构（由字母 E 表示），那么 RISC-V 架构包含 16 个通用整数寄存器，由代号 x0～x15 表示。其中通用整数寄存器 x0 被预留为常数 0，其他 15 个（x1～x15）为普通的通用整数寄存器。

嵌入式架构只能是 32 位架构（由 RV32E 表示），因此每个寄存器的宽度为 32 位。

- 如果支持单精度浮点指令（由字母 F 表示）或者双精度浮点指令（由字母 D 表示），则需要另外增加一组独立的通用浮点寄存器组，包含 32 个通用浮点寄存器，标号为 f0～f31。有关通用浮点寄存器，附录在 A14.4 节的浮点指令部分将予以详述。

在汇编语言中，通用寄存器组中的每个寄存器均有别名，如图 A-1 所示。

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

图 A-1 RISC-V 通用寄存器别名

A4.2 CSR 寄存器

RISC-V 的架构中定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用专有的 12 位地址编码空间。请参见附录 B 了解 CSR 寄存器的列表与详细信息。

A5 指令 PC

指令 PC（Instruction Program Counter）是指令存放于存储器中的地址位置。

在一部分处理器架构中，当前执行指令的 PC 值可以被反映在某些通用寄存器或特殊寄存器中。但是在 RISC-V 架构中，当前执行指令的 PC 值，并没有被反映在任何寄存器中。程序若想读取 PC 的值，只能通过某些指令间接获得，譬如 AUIPC 指令。请参见附录 A14.2

节了解 AUIPC 指令的详情。

A6 寻址空间划分

RISC-V 架构定义了两套寻址空间。

- 数据与指令寻址空间：RISC-V 架构使用统一的地址空间，寻址空间大小取决于通用寄存器的宽度。譬如，对于 32 位的 RISC-V 架构，指令和数据寻址空间为 2 的 32 次方，即 4GB 空间。
- CSR 寻址空间：CSR 寄存器是处理器核内部的寄存器，使用其专有的 12 位地址编码空间。请参见附录 B 了解 CSR 寄存器的列表与地址分配信息。

A7 大端格式或小端格式

由于现在的主流应用是小端格式（Little-Endian），因此 RISC-V 架构仅支持小端格式。有关小端格式和大端格式的定义和区别，本书在此不做赘述，请读者自行查阅学习。

A8 工作模式

如图 A-2 所示，RISC-V 架构定义了 3 种工作模式，又称特权模式（Privileged Mode）。

- Machine Mode：机器模式，简称 M Mode。
- Supervisor Mode：监督模式，简称 S Mode。
- User Mode：用户模式，简称 U Mode。

RISC-V 架构定义 M Mode 为必选模式，另外两种为可选模式。如图 A-3 所示，通过不同的模式组合可以实现不同的系统。

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

图 A-2 RISC-V 的 3 种特权模式

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

图 A-3 RISC-V 不同特权模式的组合

- 仅有机器模式一种的系统，通常为简单的嵌入式系统。
- 支持机器模式与用户模式的系统，此类系统可以实现用户和机器模式的区分，从而

实现资源保护。

- 支持机器模式、监督模式与用户模式的系统，此类系统可以实现类似 Unix 的操作系统。

A9 Hart 概念

由于现今的处理器设计技术突飞猛进，早已突破了多核的概念，甚至在一个处理器核中设计多个硬件线程的技术也早已成熟。譬如硬件超线程（Hyper-threading）技术，便是将一个处理器核中实现多份硬件线程，每套线程有自己独立的寄存器组等上下文资源，但是大多数的运算资源均被所有硬件线程复用，因此面积效率很高。在这样的硬件超线程处理器中，一个核内便存在着多个硬件线程（Hardware Thread）。

基于上述原因，在某些场景下，笼统地使用“处理器核”概念进行描述会有失精确。因此在 RISC-V 的架构文档中严谨地定义了一个 Hart（取“Hardware Thread”之意）的概念，表示一个硬件线程。在本书对于指令集架构的介绍中，将会多次使用 Hart 概念。

以蜂鸟 E200 处理器核的实现为例，由于蜂鸟 E200 是单核处理器，且没有实现任何硬件超线程的技术，因此一个蜂鸟 E200 处理器核即为一个 Hart。

A10 复位状态

对于硬件上电复位（Reset）后的行为，RISC-V 架构规定如下。

- 特权模式复位成为机器模式。
- mstatus 寄存器中的 MIE 和 MPRV 域被复位为 0 值，请参见附录 B2.9 节了解 mstatus 寄存器域的详细信息。
- PC 的复位值由硬件实现自定义，RISC-V 架构并未强制规定。
- 如果硬件实现需要区分不同的复位类型，那么 mcause 寄存器的值被复位成硬件实现自定义的值；如果硬件实现不需要区分不同的复位类型，那么 mcause 寄存器的值应该复位成为 0 值。
- 除上述寄存器之外的所有其他寄存器，RISC-V 架构并未强制规定其复位值。

A11 中断和异常

请参见第 13 章，系统了解中断和异常的相关信息。

A12 存储器地址管理

RISC-V 架构可以支持几种对存储器地址的管理模式，包括对物理地址和虚拟地址的管理方法，使得 RISC-V 架构既能支持简单的嵌入式系统（直接操作物理地址），也能支持复杂的操作系统（直接操作虚拟地址）。

由于此内容超出了本书的介绍范围（蜂鸟 E200 没有实现 MPU 或者 MMU），因此在此不做过多介绍。感兴趣的读者请参见 RISC-V“特权架构文档”原文。

A13 存储器模型

本节介绍 RISC-V 架构的存储器模型。在 RISC-V 的“指令集文档”中并未对存储器模型概念进行系统解释，原因在于“指令集文档”是对 RISC-V 架构的精确定义，而非计算机体系结构的教学文章。

为了便于读者理解，本书单独设立附录 D，对存储器模型的相关知识背景予以简介。同时，存储器模型是计算机体系结构中一个非常晦涩的概念，本书虽然力求行文通俗，但是对于此概念的阐述仍相对其他章节更为难以理解。对于初学者而言，作者建议将此节放到最后来学习。

阅读了附录 D 的读者，应该已经了解松散一致性模型（Relaxed Consistency Model）的概念以及 RISC-V 架构中定义的 Hart 概念。RISC-V 架构明确规定在不同 Hart 之间使用松散一致性模型，并相应地定义了存储器屏障指令（FENCE 和 FENCE.I）用于屏障存储器访问的顺序。另外，RISC-V 架构定义了可选的（非必需的）存储器原子操作指令（A 扩展指令子集），可进一步支持松散一致性模型。

A14 指令类型

A14.1 RV32IMAFDC 指令列表

附录仅对 RV32IMAFDC 架构所涉及的指令子集进行介绍。RV32IMAFDC 的完整指令列表及其编码请参见附录 F。

A14.2 基本整数指令（RV32I）

1. 整数有符号数

注意：RISC-V 架构中规定的所有整数有符号数均由二进制补码表示。

2. 整数运算指令

ADDI，SLTI，SLTIU，ANDI，ORI，XORI，SLLI，SRLI，SRAI 指令

（1）指令汇编格式

```
addi    rd, rs1, imm[11:0]
slti    rd, rs1, imm[11:0]
sltiu   rd, rs1, imm[11:0]
andi    rd, rs1, imm[11:0]
ori     rd, rs1, imm[11:0]
xori    rd, rs1, imm[11:0]
slli    rd, rs1, shamt[4:0]
srli    rd, rs1, shamt[4:0]
srai    rd, rs1, shamt[4:0]
```

（2）指令详解

该组指令将寄存器与立即数进行基本的整数运算操作。

- addi 指令将操作数寄存器 rs1 中的整数值与 12 位立即数（进行符号位扩展）进行加法操作，结果写回寄存器 rd 中。如果发生了结果溢出，无须特殊处理，将溢出位舍弃，仅保留低 32 位结果。

  使用“ADDI rd, rs1, 0”等效于伪指令“MV rd, rs1”，使用“ADDI x0, x0, 0”等效于伪指令“NOP”，请参见附录 G 了解更多伪指令信息。
- slti 指令将操作数寄存器 rs1 中的整数值与 12 位立即数（进行符号位扩展）当作有符号数进行比较。如果 rs1 中的值小于立即数的值，则结果为 1，否则为 0，结果写回寄存器 rd 中。
- sltiu 指令将操作数寄存器 rs1 中的整数值与 12 位立即数（仍然进行符号位扩展）当作无符号数进行比较。如果 rs1 中的值小于立即数的值，则结果为 1，否则为 0，结果写回寄存器 rd 中。

  使用“SLTIU rd, rs1, 1”等效于伪指令“SEQZ rd, rs1”，请参见附录 G 了解更多伪指令信息。

  注意：此指令的比较操作虽然是将操作数当作无符号数进行比较，但是立即数仍然是进行符号位扩展。
- andi 指令将操作数寄存器 rs1 中的整数值与 12 位立即数（进行符号位扩展）进行与（AND）操作，结果写回寄存器 rd 中。
- ori 指令将操作数寄存器 rs1 中的整数值与 12 位立即数（进行符号位扩展）进行或

（OR）操作，结果写回寄存器 rd 中。

- xori 指令将操作数寄存器 rs1 中的整数值与 12 位立即数（进行符号位扩展）进行异或（XOR）操作，结果写回寄存器 rd 中。
  使用“XORI rd, rs1, -1”等效于伪指令“NOT rd, rs1”，请参见附录 G 了解更多伪指令信息。
- slli 指令对操作数寄存器 rs1 中的整数值进行逻辑左移运算（低位补入 0），移位量为 5 位立即数，结果写回寄存器 rd 中。
- srli 指令对操作数寄存器 rs1 中的整数值进行逻辑右移运算（高位补入 0），移位量为 5 位立即数，结果写回寄存器 rd 中。
- srai 指令对操作数寄存器 rs1 中的整数值进行算术右移运算（高位补入符号位），移位量为 5 位立即数，结果写回寄存器 rd 中。

LUI，AUIPC 指令

（1）指令汇编格式

```
lui      rd, imm
auipc    rd, imm
```

（2）指令详解

- lui 指令将 20 位立即数的值左移 12 位（低 12 位补 0）成为一个 32 位数，将此数写回寄存器 rd 中。
- auipc 指令将 20 位立即数的值左移 12 位（低 12 位补 0）成为一个 32 位数，将此数与该指令的 PC 值相加，将加法结果写回寄存器 rd 中。

ADD，SUB，SLT，SLTU，AND，OR，XOR，SLL，SRL，SRA 指令

（1）指令汇编格式

```
add     rd, rs1, rs2
sub     rd, rs1, rs2
slt     rd, rs1, rs2
sltu    rd, rs1, rs2
and     rd, rs1, rs2
or      rd, rs1, rs2
xor     rd, rs1, rs2
sll     rd, rs1, rs2
srl     rd, rs1, rs2
sra     rd, rs1, rs2
```

（2）指令详解

该组指令将寄存器与寄存器进行基本的整数运算操作。

- add 指令将操作数寄存器 rs1 中的整数值与寄存器 rs2 中的整数值进行加法操作，结果写回寄存器 rd 中。如果发生了结果溢出，无须特殊处理，将溢出位舍弃，仅保留低 32 位结果。
- sub 指令将操作数寄存器 rs1 中的整数值与寄存器 rs2 中的整数值进行减法操作，结

果写回寄存器 rd 中。如果发生了结果溢出，无须特殊处理，将溢出位舍弃，仅保留低 32 位结果。

- slt 指令将操作数寄存器 rs1 中的整数值与寄存器 rs2 中的整数值当作有符号数进行比较。如果 rs1 中的值小于 rs2 中的值，则结果为 1，否则为 0，结果写回寄存器 rd 中。
- sltu 指令将操作数寄存器 rs1 中的整数值与寄存器 rs2 中的整数值当作无符号数进行比较。如果 rs1 中的值小于 rs2 中的值，则结果为 1，否则为 0，结果写回寄存器 rd 中。
- and 指令将操作数寄存器 rs1 中的整数值与寄存器 rs2 中的整数值进行与（AND）操作，结果写回寄存器 rd 中。
- or 指令将操作数寄存器 rs1 中的整数值与寄存器 rs2 中的整数值进行或（OR）操作，结果写回寄存器 rd 中。
- xor 指令将操作数寄存器 rs1 中的整数值与寄存器 rs2 中的整数值进行异或（XOR）操作，结果写回寄存器 rd 中。
- sll 指令对操作数寄存器 rs1 中的整数值进行逻辑左移运算（低位补入 0），移位量为寄存器 rs2 中整数值的低 5 位，结果写回寄存器 rd 中。
- srl 指令对操作数寄存器 rs1 中的整数值进行逻辑右移运算（高位补入 0），移位量为寄存器 rs2 中整数值的低 5 位，结果写回寄存器 rd 中。
- sra 指令对操作数寄存器 rs1 中的整数值进行算术右移运算（高位补入符号位），移位量为寄存器 rs2 中整数值的低 5 位，结果写回寄存器 rd 中。

3. 分支跳转指令

请先参见第 7.1.4 节了解更多分支跳转指令的知识背景和使用信息。

JAL，JALR 指令

（1）指令汇编格式

```
jal    rd, label
jalr   rd, rs1, imm
```

（2）指令详解

该组指令为无条件跳转指令，即一定会发生跳转：

- jal 指令使用 20 位立即数（有符号数）作为偏移量（offset）。该偏移量乘以 2，然后与该指令的 PC 相加，生成得到最终的跳转目标地址，因此仅可以跳转到前后 1MB 的地址区间。jal 指令将其下一条指令的 PC（即当前指令 PC+4）的值写入其结果寄存器 rd 中。

  注意： 在实际的汇编程序编写中，跳转的目标往往使用汇编程序中的 label，汇编器会自动根据 label 所在的地址计算出相对的偏移量赋予指令编码。
- jalr 指令使用 12 位立即数（有符号数）作为偏移量，与操作数寄存器 rs1 中的值相加得到最终的跳转目标地址。jalr 指令将其下一条指令的 PC（即当前指令 PC+4）的值写入其结果寄存器 rd。

BEQ，BNE，BLT，BLTU，BGE，BGEU 指令

（1）指令汇编格式

```
beq     rs1, rs2, label
bne     rs1, rs2, label
blt     rs1, rs2, label
bltu    rs1, rs2, label
bge     rs1, rs2, label
bgeu    rs1, rs2, label
```

（2）指令详解

该组指令为有条件跳转指令，使用 12 位立即数（有符号数）作为偏移量。该偏移量乘以 2，然后与该指令的 PC 相加，生成得到最终的跳转目标地址，因此仅可以跳转到前后 4KB 的地址区间。有条件跳转指令需要在条件为真时才会发生跳转，具体如下。

- beq 指令只有在操作数寄存器 rs1 中的数值与操作数寄存器 rs2 中的数值相等时，才会跳转。
- bne 指令只有在操作数寄存器 rs1 中的数值与操作数寄存器 rs2 中的数值不相等时，才会跳转。
- blt 指令只有在操作数寄存器 rs1 中的有符号数小于操作数寄存器 rs2 中的有符号数时，才会跳转。
- bltu 指令只有在操作数寄存器 rs1 中的无符号数小于操作数寄存器 rs2 中的无符号数时，才会跳转。
- bge 指令只有在操作数寄存器 rs1 中的有符号数大于或等于操作数寄存器 rs2 中的有符号数时，才会跳转。
- bgeu 指令只有在操作数寄存器 rs1 中的无符号数大于或等于操作数寄存器 rs2 中的无符号数时，才会跳转。

注意：在实际的汇编程序编写中，跳转的目标往往使用汇编程序中的 label，汇编器会自动根据 label 所在的地址计算出相对的偏移量赋予指令编码。

4．整数 Load/Store 指令

LW，LH，LHU，LB，LBU，SW，SH，SB 指令

（1）指令汇编格式

```
lw    rd, offset[11:0](rs1)
lh    rd, offset[11:0](rs1)
lhu   rd, offset[11:0](rs1)
lb    rd, offset[11:0](rs1)
lbu   rd, offset[11:0](rs1)
sw    rs2, offset[11:0](rs1)
sh    rs2, offset[11:0](rs1)
sb    rs2, offset[11:0](rs1)
```

（2）指令详解

该组指令进行存储器读或者写操作，访问存储器的地址均由操作数寄存器 rs1 中的值与 12 位的立即数（进行符号位扩展）相加所得。

- lw 指令从存储器中读回一个 32 位的数据，写回寄存器 rd 中。
- lh 指令从存储器中读回一个 16 位的数据，进行符号位扩展后写回寄存器 rd 中。
- lhu 指令从存储器中读回一个 16 位的数据，进行高位补 0 扩展后写回寄存器 rd 中。
- lb 指令从存储器中读回一个 8 位的数据，进行符号位扩展后写回寄存器 rd 中。
- lbu 指令从存储器中读回一个 8 位的数据，进行高位补 0 扩展后写回寄存器 rd 中。
- sw 指令将操作数寄存器 rs2 中的 32 位数据，写回存储器中。
- sh 指令将操作数寄存器 rs2 中的低 16 位数据，写回存储器中。
- sb 指令将操作数寄存器 rs2 中的低 8 位数据，写回存储器中。

对于整数 Load 和 Store 指令，RISC-V 架构推荐使用地址对齐的存储器读写操作。但是地址非对齐的存储器操作 RISC-V 架构也支持，处理器可以选择用硬件来支持，也可以选择用软件异常服务程序来支持。蜂鸟 E200 处理器核选择采用软件异常服务程序来支持（即地址非对齐的 Load 或 Store 指令会产生异常），参见第 13 章了解更多异常的相关信息。

注意：RISC-V 架构仅支持小端（Little-Endian）格式。

对于地址对齐的存储器读写操作，RISC-V 架构规定其存储器读写操作必须具备原子性。有关存储器原子操作的背景知识请参见附录 A14.5 节对于 RV32A 指令子集的介绍。

5．CSR 指令

如附录 A4.2 节所述，RISC-V 的架构中定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用其专有的 12 位地址编码空间。请参见附录 B 了解 CSR 寄存器的列表与详细信息。

CSR 寄存器的访问采用专用的 CSR 指令，包括 CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI 以及 CSRRCI 指令。

CSRRW，CSRRS，CSRRC，CSRRWI，CSRRSI，CSRRCI 指令

（1）指令汇编格式

```
csrrw    rd, csr, rs1
csrrs    rd, csr, rs1
csrrc    rd, csr, rs1
csrrwi   rd, csr, imm[4:0]
csrrsi   rd, csr, imm[4:0]
csrrci   rd, csr, imm[4:0]
```

（2）指令详解

该组指令用于读写 CSR 寄存器。

- csrrw 指令完成两项操作：

将 csr 索引的 CSR 寄存器值读出，写回结果寄存器 rd 中。

将操作数寄存器 rs1 中的值写入 csr 索引的 CSR 寄存器中。

- csrrs 指令完成两项操作：

  将 csr 索引的 CSR 寄存器值读出，写回结果寄存器 rd 中。

  以操作数寄存器 rs1 中的值逐位作为参考，如果 rs1 中的值某个比特位为 1，则将 csr 索引的 CSR 寄存器中对应的比特位置为 1，其他位则不受影响。

- csrrc 指令完成两项操作：

  将 csr 索引的 CSR 寄存器的值读出，写回结果寄存器 rd 中。

  以操作数寄存器 rs1 中的值逐位作为参考，如果 rs1 中的值某个比特位为 1，则将 csr 索引的 CSR 寄存器中对应的比特位清为 0，其他位则不受影响。

- csrrwi 指令完成两项操作：

  将 csr 索引的 CSR 寄存器的值读出，写回结果寄存器 rd 中。

  将 5 位立即数（高位补 0 扩展）的值写入 csr 索引的 CSR 寄存器中。

- csrrsi 指令完成两项操作：

  将 csr 索引的 CSR 寄存器的值读出，写回结果寄存器 rd 中。

  以 5 位立即数（高位补 0 扩展）的值逐位作为参考，如果 rs1 中的值某个比特位为 1，将 csr 索引的 CSR 寄存器中对应的比特位置为 1，其他位则不受影响。

- csrrci 指令完成两项操作：

  将 csr 索引的 CSR 寄存器的值读出，写回结果寄存器 rd 中。

  以 5 位立即数（高位补 0 扩展）的值逐位作为参考，如果 rs1 中的值某个比特位为 1，将 csr 索引的 CSR 寄存器中对应的比特位清为 0，其他位则不受影响。

注意：

- 对于 CSRRW 和 CSRRWI 指令而言，如果结果寄存器 rd 的索引值为 0，则不会发起 CSR 寄存器的读操作，也不会带来任何读操作造成的副作用。
- 对于 CSRRS 和 CSRRC 指令而言，如果 rs1 的索引值为 0，则不会发起 CSR 寄存器的写操作，也不会带来任何写操作造成的副作用。
- 对于 CSRRSI 和 CSRRCI 指令而言，如果立即数的值为 0，则不会发起 CSR 寄存器的写操作，也不会带来任何写操作造成的副作用。

使用上述指令的不同形式可以等效出 CSRR、CSRW、CSRS 以及 CSRC 等伪指令，如请参见附录 G 了解更多伪指令信息。

6．存储器屏障（FENCE）指令

在附录 A13 中已介绍，RISC-V 架构在不同 Hart 之间使用的是松散一致性模型，也介绍了松散一致性模型需要使用存储器屏障（Memory Fence）指令，因此 RISC-V 也相应地定义

了其存储器屏障指令，主要包括 FENCE 和 FENCE.I 指令，此二种指令都是 RISC-V 架构必选的基本指令。

FENCE 指令

（1）指令汇编格式

```
fence
```

（2）指令详解

fence 指令用于屏障“数据”存储器访问的执行顺序，在程序中如果添加了一条 fence 指令，则该 fence 指令能够保证“在 fence 之前所有指令进行的数据访存结果”必须比“在 fence 之后所有指令进行的数据访存结果”先被观测到。通俗地讲，fence 指令就像一堵屏障一样，在 fence 指令之前的所有数据存储器访问指令，必须比该 fence 指令之后的所有数据存储器访问指令先执行。

为了能够更加细致地屏障不同地址区间的存储器访问指令，RISC-V 架构将数据存储器的地址空间分为设备 I/O（Device I/O）和普通存储器（Memory）空间，因此对其读写访问可以分为 4 种类型。

- I：设备读（Device-Input）
- O：设备写（Device-Output）
- R：存储器读（Memory-Reads）
- W：存储器写（Memory-Writes）

如图 A-4 所示，fence 指令的编码中包含了 PI/PO/PR/PW 编码位，分别表示 fence 指令之前（predecessor）的四种读写访问类型；还包含了 SI/SO/SR/SW 编码位，分别表示 fence 指令之后（successor）的四种读写访问类型。通过设置不同的编码位，就可以更加细致地屏障不同数据存储器访问操作。譬如，在程序中如果添加了一条“fence io, iorw”指令，则该 FENCE 指令能够保证“在 fence 之前所有指令进行的设备读（Device-Input）和设备写（Device-Output）操作结果”必须比“在 fence 之后所有指令进行的设备读（Device-Input）、设备写（Device-Output）、存储器读（Memory-Reads）以及存储器写（Memory-Writes）结果”先被观测到。通俗地讲，fence 指令就像一堵屏障一样，在 fence 指令之前的（Device-Input）和设备写（Device-Output）操作指令，必须比该 fence 指令之后的设备读（Device-Input）、设备写（Device-Output）、存储器读（Memory-Reads）以及存储器写（Memory-Writes）操作指令先执行。

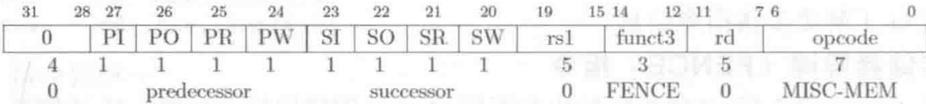


图 A-4 Fence 指令的指令编码

注意：不带参数的 fence 指令默认等效于“fence iorw, iorw”。虽然 fence 指令可以通过 IORW 参数细致地屏障不同地址类型的存储器访问指令，但是协议也允许处理器的简单硬件实现，譬如对于简单的低功耗处理器而言，不管 fence 指令中编码的 PI/PO/PR/PW/SI/SO/SR/RW 的值如何，一概的屏障所有地址类型的存储器访问指令（都等效于 FENCE IORW, IORW），蜂鸟 E200 处理器核便是采取这种简单的硬件实现。

FENCE.I 指令

（1）指令汇编格式

```
fence.i
```

（2）指令详解

fence.i 指令用于同步指令和数据流。

为了能够解释清楚 fence.i 指令的功能。在此有必要先引出一个问题：假设在程序中一条“写存储器指令”向某段地址区间中写入了新的值，同时，假设“后序的指令”也需要从该地址区间进行取指令，那么该取指操作能否取到它前面“写存储器指令”写入的新值呢？答案是“不一定”。因为结合第 5 章对于流水线的介绍便可以知道，处理器的流水线具有一定的深度，指令与指令采取的是流水线的方式工作，当“写存储器指令”完成了写操作之时，后续的指令可能早已完成了取指令操作进入了流水线的执行阶段，因此“后序的指令”取指取到的其实是它前面“写存储器指令”写入新值之前的旧值。

为了解决该问题，fence.i 指令被引入。如果在程序中如果添加了一条 fence.i 指令，则该 fence.i 指令能够保证“在 fence.i 之前所有指令进行的数据访存结果”一定能够被“在 fence.i 之后所有指令进行的取指令操作”访问到。通常说来，在处理器的微架构硬件实现时，一旦遇到一条 fence.i 指令便会先等待之前的所有的数据访存指令执行完，然后将流水线冲刷掉（包括 I-Cache），使其后续的所有指令能够重新进行取指，从而取到最新的值。

注意：fence.i 指令只能够保证同一个 Hart 执行的指令和数据流顺序，而无法保证多个 Hart 之间的指令和数据流顺序。假设一个 Hart 希望其执行的数据访存结果能够被所有 Hart（包括其自己和其他 Hart）的指令取指操作所访问到，那么理论上它应该采取如下措施。

- 第 1 步：本 Hart 完成“数据访存操作”。
- 第 2 步：本 Hart 执行一条 fence 指令，保证其前序的所有数据访存操作一定能够比后序的操作被所有的 Hart 先观测到。
- 第 3 步：本 Hart 请求所有的 Hart（包括其自己）执行一条 fence.i 指令。
- 注意：本 Hart“对其他的 Hart 发起的请求操作”和之前进行的“数据访存操作”必须能够被 fence 指令屏障开，也就意味着，当所有其他 Hart 接收到请求之后，一定能够观测到之前“数据访存操作”的结果，然后再执行了 fence.i 指令之后的指令取指操作便能够取到最新的数值。

7. 特殊指令 ECALL、EBREAK、MRET、WFI

ECALL 指令

（1）指令汇编格式

```
ecall
```

（2）指令详解

ecall 指令用于生成环境调用（Environment-Call）异常。当产生异常时，mepc 寄存器将会被更新为 ecall 指令本身的 PC 值。请参见第 13 章了解更多中断与和异常信息。

EBREAK 指令

（1）指令汇编格式

```
ebreak
```

（2）指令详解

ebreak 指令用于生成断点（Breakpoint）异常。当产生异常时，mepc 寄存器将会被更新为 ebreak 指令本身的 PC 值。请参见第 13 章了解更多中断与和异常信息。

MRET 指令

（1）指令汇编格式

```
mret
```

（2）指令详解

RISC-V 架构定义了一组专门用于退出异常的指令，称之为异常返回指令（Trap-Return Instructions），包括 mret、sret 和 uret，其中 mret 指令是必备的，而 sret 和 uret 指令仅在支持监督模式和用户模式的处理器中使用。使用 mret 指令退出异常的机制如下。

- 处理器在执行了 mret 指令退出异常时，即跳转到 mepc 寄存器的值指定的 PC 地址。由于在之前进入异常时，mepc 寄存器被同时更新以反映当时遇到异常的指令的 PC 值，因此通过这个机制则意味着 mret 指令执行后处理器回到了当时遇到异常的指令的 PC 地址，从而可以继续执行之前被中止的程序流。
- 处理器在执行了 mret 指令退出异常时，mstatus 寄存器的有些域被同时更新。
  MIE 的值被更新为 MPIE 的值，MPIE 的值则被更新为 1。
  假设只支持机器模式，MPP 的值永远为 11。
  由于在进入异常时，MPIE 的值曾经被更新为 MIE 的值（MIE 的值则更新为 0 以全局关闭中断），因此通过这个机制则意味着 mret 指令执行后处理器的 MIE 值被更新回了之前的值（假设之前的 MIE 值为 1，则意味着中断被重新全局打开）。

请参见第 13 章了解更多中断与和异常信息。

WFI 指令

（1）指令汇编格式

```
wfi
```

（2）指令详解

WFI 指令，全称为等待中断（Wait For Interrupt），是 RISC-V 架构定义的专门用于休眠的指令。

RISC-V 架构也允许具体的硬件实现中将 WFI 指令当成一种 NOP 操作，即什么也不做。如果硬件实现选择支持休眠模式，则按照 RISC-V 架构规定其行为如下。

- 当处理器执行到 WFI 指令之后，将会停止执行当前的指令流，进入一种空闲状态，这种空闲状态可以被称之为“休眠”状态。
- 直到处理器接收到中断（中断局部开关必须被打开，由 mie 寄存器控制），处理器便被唤醒。处理器被唤醒后，如果中断被全局打开（mstatus 寄存器的 MIE 域控制），则进入到中断异常服务程序开始执行；如果中断被全局关闭，则继续顺序执行之前停止的指令流。

A14.3 整数乘法和除法指令（RV32M 指令子集）

RISC-V 架构定义了可选的整数乘法和除法指令（M 扩展指令子集）。本书仅介绍 32 位架构的乘除法指令（RV32M）。

1. 整数乘法指令

MUL，MULH，MULHU，MULHSU 指令

（1）指令汇编格式

```
mul      rd, rs1, rs2
mulh     rd, rs1, rs2
mulhu    rd, rs1, rs2
mulhsu   rd, rs1, rs2
```

（2）指令详解

该组指令进行整数的乘法操作。

- mul 指令将操作数寄存器 rs1 与 rs2 中的 32 位整数相乘，将结果的低 32 位写回寄存器 rd 中。
- 由于两个 32 位整数操作数相乘的结果等于 64 位，且对于两个 32 位整数相乘而言，将两个操作数当作有符号数相乘所得的低 32 位和当作无符号数相乘所得的低 32 位肯定是相同的（具体算法读者可以自行推导），因此 RISC-V 架构仅定义了一条 mul 指令作为取低 32 位结果的乘法指令。
- mulh 指令将操作数寄存器 rs1 与 rs2 中的 32 位整数相乘，其中 rs1 和 rs2 中的值都

被当作有符号数，将结果的高 32 位写回寄存器 rd 中。

- mulhu 指令将操作数寄存器 rs1 与 rs2 中的 32 位整数相乘，其中 rs1 和 rs2 中的值都被当作无符号数，将结果的高 32 位写回寄存器 rd 中。
- mulhsu 指令将操作数寄存器 rs1 与 rs2 中的 32 位整数相乘，其中 rs1 和 rs2 中的值分别被当作有符号数和无符号数，将结果的高 32 位写回寄存器 rd 中。

注意：如果希望得到两个 32 位整数相乘的完整 64 位结果，RISC-V 架构推荐使用两条连续的乘法指令"**MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1,rs2**"，其要点如下。

- 两条指令的源操作数索引号和顺序必须完全相同。
- 第一条指令的结果寄存器 rdh 的索引号必须不能与其 rs1 和 rs2 的索引号相等。
- 处理器实现的微架构可以将两条指令融合（Fused）成为一条指令执行，而不是分离的两条指令，从而提高性能。

2. 整数除法指令

DIV，DIVU，REM，REMU 指令

（1）指令汇编格式

```
div     rd, rs1, rs2
divu    rd, rs1, rs2
rem     rd, rs1, rs2
remu    rd, rs1, rs2
```

（2）指令详解

该组指令进行整数的除法操作。

- div 指令将操作数寄存器 rs1 与 rs2 中的 32 位整数相除，其中 rs1 和 rs2 中的值都被当作有符号数，将除法所得的商写回寄存器 rd 中。
- divu 指令将操作数寄存器 rs1 与 rs2 中的 32 位整数相除，其中 rs1 和 rs2 中的值都被当作无符号数，将除法所得的商写回寄存器 rd 中。
- rem 指令将操作数寄存器 rs1 与 rs2 中的 32 位整数相除，其中 rs1 和 rs2 中的值都被当作有符号数，将除法所得的余数写回寄存器 rd 中。
- remu 指令将操作数寄存器 rs1 与 rs2 中的 32 位整数相除，其中 rs1 和 rs2 中的值都被当作无符号数，将除法所得的余数写回寄存器 rd 中。

注意：如果希望同时得到两个 32 位整数相除的商和余数，RISC-V 架构推荐使用两条连续的除法和取余指令"**DIV[U] rdq, rs1, rs2; REM[U] rdr, rs1, rs2**"，其要点如下。

- 两条指令的源操作数索引号和顺序必须完全相同。
- 第一条指令的结果寄存器 rdq 的索引号必须不能与其 rs1 和 rs2 的索引号相等。
- 处理器实现的微架构可以将两条指令融合成为一条指令执行，而不是分离的两条指令，从而提高性能。

在很多的处理器架构中，除法的除以 0（Divided-by-Zero）都会触发异常跳转（Trap）

从而进入异常模式。但是请注意：RISC-V 架构的除法指令在除以 0 时并不会跳转进入异常模式。这是 RISC-V 架构的一个显著特点。该特点可以大幅简化处理器流水线的硬件实现。

虽然不会发生异常，但是仍然会产生特殊的数值结果。RISC-V 架构的除法指令在除以 0，以及发生结果上溢出时产生的数值结果如图 A-5 所示。

Condition	Dividend	Divisor	DIVU	REMU	DIV	REM
Division by zero	x	0	$2^{XLEN}-1$	x	-1	x
Overflow (signed only)	-2^{XLEN-1}	-1	–	–	-2^{XLEN-1}	0

图 A-5　DIVU/REMU/DIV/REM 四条指令在除以 0 和上溢出时的结果

A14.4　浮点指令（RV32F，RV32D 指令子集）

RISC-V 架构定义了可选的单精度浮点指令（F 扩展指令子集）和双精度浮点指令（D 扩展指令子集）。

注意：RISC-V 架构规定，处理器可以选择只实现 F 扩展指令子集而不支持 D 扩展指令子集；但是如果支持了 D 扩展指令子集，则必须支持 F 扩展指令子集。

本书仅介绍 32 位架构的浮点指令（RV32F，RV32D）。

1．标准

RISC-V 架构中规定的所有浮点运算均遵循标准 IEEE-754 标准。具体的标准版本为 ANSI/IEEE Std 754-2008, IEEE standard for oating-point arithmetic, 2008.

2．通用浮点寄存器组

RISC-V 架构规定，如果支持单精度浮点指令或者双精度浮点指令，则需要增加一组独立的通用浮点寄存器组，包含 32 个通用浮点寄存器，标号为 f0 至 f31。

浮点寄存器的宽度由 FLEN 表示，如果仅支持 F 扩展指令子集，则每个通用浮点寄存器的宽度为 32 位；如果支持 D 扩展指令子集，则每个通用浮点寄存器的宽度为 64 位。

注意：RISC-V 架构规定，不同于基本整数指令集中规定 x0 为常数 0，浮点寄存器组中的 f0 为一个正常的通用浮点寄存器（与 f1～f31 相同）。

3．浮点 fcsr 寄存器

RISC-V 架构规定，如果支持单精度浮点指令或者双精度浮点指令，则需要增加一个浮点控制状态寄存器（fcsr），如图 A-6 所示。fcsr 是一个可读可写的 CSR 寄存器，有关此 CSR 寄存器的地址，请参见附录 B1。

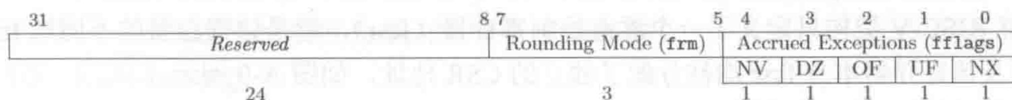


图 A-6　fcsr 寄存器格式

4. 浮点异常标志

如图 A-6 所示，fcsr 寄存器包含浮点异常标志位域（fflags），不同的异常标志位所表示的异常类型如图 A-7 所示。如果浮点运算单元在运算中出现了相应的异常，则会将 fcsr 寄存器中对应的异常标志位设置为高，且会一直保持累积。软件可以通过写 0 的方式单独清除某个异常标志位。

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

图 A-7 异常标志位

在很多的处理器架构中，浮点运算产生结果异常都会触发异常跳转（Trap）从而进入异常模式。但是请注意：RISC-V 架构的浮点指令在产生结果异常时并不会跳转进入异常模式，而是如上所述仅设置 fcsr 寄存器中的异常标志位。这是 RISC-V 架构的一个显著特点。该特点可以大幅简化处理器流水线的硬件实现。

5. 浮点舍入模式

根据 IEEE-754 标准，浮点数运算需要指定舍入模式（Rounding Mode），RISC-V 架构浮点运算的舍入模式可以通过两种方式指定。

（1）静态舍入模式：浮点指令的编码中有 3 位作为舍入模式域，有关浮点指令列表以及指令编码请参见附录 F4。不同的舍入模式编码如图 A-8 所示，RISC-V 架构支持五种合法的舍入模式。除此之外，如果舍入模式编码为 101 或 110，则为非法模式；如果舍入模式编码为 111，则意味着使用动态舍入模式。

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Invalid. Reserved for future use.*
110		*Invalid. Reserved for future use.*
111		In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid.*

图 A-8 舍入模式位

（2）动态舍入模式：如果使用动态舍入模式，则使用 fcsr 寄存器中的舍入模式域。如图 A-6 所示，fcsr 寄存器包含舍入模式域。不同的舍入模式编码同样如图 A-8 所示，仅支持五种合法的舍入模式。如果 fcsr 寄存器中的舍入模式域指定为非法的舍入模式，则后续浮点指令会产生非法指令异常。

6. 浮点 fcsr 访问伪指令

虽然 RISC-V 架构只定义了一个浮点控制寄存器（fcsr），但是该寄存器的不同域 frm 和 fflags 以及该寄存器本身 fcsr 均被分配了独立的 CSR 地址，如图 A-9 所示。

为了能够方便地访问以上浮点 CSR 寄存器，RISC-V 架构定义了一系列的伪指令，如图 A-10 所示。所谓伪指令意味着其并不是一条真正的指令，而是对其他基本指令使用形式

的一种别名，譬如伪指令“frcsr rd”事实上是基本 CSR 指令的使用形式“csrrs rd, fcsr, x0”。

Number	Privilege	Name	Description
Floating-Point Control and Status Registers			
0x001	Read/write	fflags	Floating-Point Accrued Exceptions.
0x002	Read/write	frm	Floating-Point Dynamic Rounding Mode.
0x003	Read/write	fcsr	Floating-Point Control and Status Register (frm + fflags).

图 A-9　fflags，frm 和 fcsr 的 CSR 地址

frcsr rd	csrrs rd, fcsr, x0	Read FP control/status register
fscsr rd, rs	csrrw rd, fcsr, rs	Swap FP control/status register
fscsr rs	csrrw x0, fcsr, rs	Write FP control/status register
frrm rd	csrrs rd, frm, x0	Read FP rounding mode
fsrm rd, rs	csrrw rd, frm, rs	Swap FP rounding mode
fsrm rs	csrrw x0, frm, rs	Write FP rounding mode
fsrmi rd, imm	csrrwi rd, frm, imm	Swap FP rounding mode, immediate
fsrmi imm	csrrwi x0, frm, imm	Write FP rounding mode, immediate
frflags rd	csrrs rd, fflags, x0	Read FP exception flags
fsflags rd, rs	csrrw rd, fflags, rs	Swap FP exception flags
fsflags rs	csrrw x0, fflags, rs	Write FP exception flags
fsflagsi rd, imm	csrrwi rd, fflags, imm	Swap FP exception flags, immediate
fsflagsi imm	csrrwi x0, fflags, imm	Write FP exception flags, immediate

图 A-10　浮点 CSR 访问伪指令

7. 关闭浮点单元

如果处理器不想使用浮点运算单元（譬如将浮点单元关电以节省功耗），可以使用 CSR 写指令将 mstatus 寄存器的 FS 域设置成 0，将浮点单元的功能予以关闭。当浮点单元的功能关闭之后，任何访问浮点 CSR 寄存器的操作或者任何执行浮点指令的行为都将会产生非法指令（Illegal Instruction）异常。

请参见附录 B2.9 节了解 mstatus 寄存器 FS 域的相关信息。

8. 非规格化数的处理

RISC-V 架构规定，对于非规格化数（Subnormal Numbers）的处理完全遵循附录 A14.4 节列举的 IEEE-754 标准定义。

9. Canonical-NaN 数

根据 IEEE-754 标准，在浮点数的表示中，有一类特殊编码数据属于 NaN（Not a Number）类型，且 NaN 分为 Signaling-NaN 和 Quiet-NaN。有关 NaN 数据的细节请参见附录 A14.4 节列举的 IEEE-754 标准。

RISC-V 架构规定，如果浮点运算的结果是一个 NaN 数，那么使用一个固定的 NaN 数，将之命名为 Canonical-NaN。单精度浮点对应的 Canonical-NaN 数值为 0x7fc00000，双精度浮点对应的 Canonical-NaN 数值为 0x7ff80000_00000000。

10. NaN-boxing

如果同时支持单精度浮点（F 扩展指令子集）和双精度浮点（D 扩展指令子集），由于浮点通用寄存器的宽度为 64 位，RISC-V 架构规定单精度浮点指令产生的 32 位结果写入浮

点通用寄存器（64 位宽）时，将结果写入低 32 位，而高位则全部写为数值 1，RISC-V 架构规定此种做法称之为 NaN-boxing。NaN-boxing 可以发生在如下情形：

- 对于单精度浮点读（Load）/写（Store）指令和传送（Move）指令（包括 FLW，FSW，FMV.W.X，FMV.X.W）。如果需要将 32 位的数值写入通用浮点寄存器，则采用 NaN-boxing 的方式；如果需要将浮点通用寄存器中的数值读出，则仅使用其低 32 位数值。
- 对于单精度浮点运算（Compute）和符号注入（Sign-injection）指令，需要判断其操作数浮点寄存器中的值是否为合法的 NaN-boxed 值（即高 32 位都为 1）。如果是，则正常使用其低 32 位；如果不是，则将此操作数当作 Canonical-NaN 来使用。
- 对于整数至单精度浮点的转换指令（譬如 FCVT.S.X），则采用 NaN-boxing 的方式写回浮点通用寄存器。对于单精度浮点至整数的转换指令（譬如 FCVT.X.S），需要判断其操作数浮点寄存器中的值是否为合法的 NaN-boxed 值（即高 32 位都为 1）。如果是，则正常使用其低 32 位；如果不是，则将此操作数当作 Canonical-NaN 来使用。

11. 浮点数读写指令

FLW，FSW，FLD，FSD 指令

（1）指令汇编格式

```
flw    rd, offset[11:0](rs1)
fsw    rs2, offset[11:0](rs1)
fld    rd, offset[11:0](rs1)
fsd    rs2, offset[11:0](rs1)
```

（2）指令详解

该组指令进行存储器读或者写操作，访问存储器的地址均由操作数寄存器 rs1 中的值与 12 位的立即数（进行符号位扩展）相加所得。

- flw 指令从存储器中读回一个单精度浮点数，写回寄存器 rd 中。
- fsw 指令将操作数寄存器 rs2 中的单精度浮点数，写回存储器中。
- fld 指令从存储器中读回一个双精度浮点数，写回寄存器 rd 中。
- fsd 指令将操作数寄存器 rs2 中的双精度浮点数，写回存储器中。

对于浮点读和写指令，RISC-V 架构推荐使用地址对齐的存储器读写操作。但是地址非对齐的存储器操作 RISC-V 架构也支持，处理器可以选择用硬件来支持，也可以选择用软件异常服务程序来支持。蜂鸟 E200 处理器核选择采用软件异常服务程序来支持（即地址非对齐的浮点数读或写指令会产生异常），参见第 13 章了解更多异常的相关信息。

对于地址对齐的存储器读写操作，RISC-V 架构规定其存储器读写操作必须具备原子性。有关存储器原子操作的背景知识请参见附录 A14.5 节对于 RV32A 指令子集的介绍。

12. 浮点数运算指令

注意：本节所有指令的浮点运算（Compute）均遵循附录 A14.4 节列举的 IEEE-754 的标

准定义。

FADD，FSUB，FMUL，FDIV，FSQRT 指令

（1）指令汇编格式

```
fadd.s    rd, rs1, rs2
fsub.s    rd, rs1, rs2
fmul.s    rd, rs1, rs2
fdiv.s    rd, rs1, rs2
fsqrt.s   rd, rs1
fadd.d    rd, rs1, rs2
fsub.d    rd, rs1, rs2
fmul.d    rd, rs1, rs2
fdiv.d    rd, rs1, rs2
fsqrt.d   rd, rs1
```

（2）指令详解

该组指令进行加、减、乘、除、求平方根操作。

- fadd.s 指令将操作数寄存器 rs1 与 rs2 中的单精度浮点数进行加法操作，结果写回寄存器 rd 中。
- fsub.s 指令将操作数寄存器 rs1 与 rs2 中的单精度浮点数进行减法操作，结果写回寄存器 rd 中。
- fmul.s 指令将操作数寄存器 rs1 与 rs2 中的单精度浮点数进行乘法操作，结果写回寄存器 rd 中。
- fdiv.s 指令将操作数寄存器 rs1 与 rs2 中的单精度浮点数进行除法操作，结果写回寄存器 rd 中。
- fsqrt.s 指令将操作数寄存器 rs1 中的单精度浮点数进行求平方根操作，结果写回寄存器 rd 中。
- fadd.d 指令将操作数寄存器 rs1 与 rs2 中的双精度浮点数进行加法操作，结果写回寄存器 rd 中。
- fsub.d 指令将操作数寄存器 rs1 与 rs2 中的双精度浮点数进行减法操作，结果写回寄存器 rd 中。
- fmul.d 指令将操作数寄存器 rs1 与 rs2 中的双精度浮点数进行乘法操作，结果写回寄存器 rd 中。
- fdiv.d 指令将操作数寄存器 rs1 与 rs2 中的双精度浮点数进行除法操作，结果写回寄存器 rd 中。
- fsqrt.d 指令将操作数寄存器 rs1 中的双精度浮点数进行求平方根操作，结果写回寄存器 rd 中。

FMIN，FMAX 指令

（1）指令汇编格式

```
fmin.s    rd, rs1, rs2
fmax.s    rd, rs1, rs2
fmin.d    rd, rs1, rs2
fmax.d    rd, rs1, rs2
```

（2）指令详解

该组指令进行取大值、取小值操作。

- fmin.s 指令将操作数寄存器 rs1 与 rs2 中的单精度浮点数进行比较操作，将数值小的一方作为结果写回寄存器 rd 中。
- fmax.s 指令将操作数寄存器 rs1 与 rs2 中的单精度浮点数进行比较操作，将数值大的一方作为结果写回寄存器 rd 中。
- fmin.d 指令将操作数寄存器 rs1 与 rs2 中的双精度浮点数进行比较操作，将数值小的一方作为结果写回寄存器 rd 中。
- fmax.d 指令将操作数寄存器 rs1 与 rs2 中的双精度浮点数进行比较操作，将数值大的一方作为结果写回寄存器 rd 中。

对于 FMAX 和 FMIN 指令，注意如下特殊情况。

- 如果指令的两个操作数都是 NaN，那么结果为 Canonical-NaN。
- 如果只有一个操作数为 NaN，则结果为非 NaN 的另外一个操作数。
- 如果任意一个操作数属于 Signaling-NaN，则需要在 fscr 寄存器中产生 NV 异常标志。
- 由于浮点数可以表示两个 0 值，分别是−0.0 和+0.0，对于 FMAX 和 FMIN 指令而言，−0.0 被认为比+0.0 小。

FMADD，FMSUB，FNMSUB，FNMADD 指令

（1）指令汇编格式

```
fmadd.s     rd, rs1, rs2, rs3
fmsub.s     rd, rs1, rs2, rs3
fnmadd.s    rd, rs1, rs2, rs3
fnmsub.s    rd, rs1, rs2, rs3
fmadd.d     rd, rs1, rs2, rs3
fmsub.d     rd, rs1, rs2, rs3
fnmadd.d    rd, rs1, rs2, rs3
fnmsub.d    rd, rs1, rs2, rs3
```

（2）指令详解

该组指令进行一体化乘累加（Fused Multiply-add）操作。

- fmadd.s 指令将操作数寄存器 rs1、rs2 与 rs3 中的单精度浮点数进行 rs1*rs2+rs3 操作，将结果写回寄存器 rd 中。
- fmsub.s 指令将操作数寄存器 rs1、rs2 与 rs3 中的单精度浮点数进行 rs1*rs2-rs3 操作，

将结果写回寄存器 rd 中。

- fnmadd.s 指令将操作数寄存器 rs1、rs2 与 rs3 中的单精度浮点数进行 - rs1*rs2-rs3 操作，将结果写回寄存器 rd 中。
- fnmsub.s 指令将操作数寄存器 rs1、rs2 与 rs3 中的单精度浮点数进行 - rs1*rs2+rs3 操作，将结果写回寄存器 rd 中。
- fmadd.d 指令将操作数寄存器 rs1、rs2 与 rs3 中的双精度浮点数进行 rs1*rs2+rs3 操作，将结果写回寄存器 rd 中。
- fmsub.d 指令将操作数寄存器 rs1、rs2 与 rs3 中的双精度浮点数进行 rs1*rs2-rs3 操作，将结果写回寄存器 rd 中。
- fnmadd.d 指令将操作数寄存器 rs1、rs2 与 rs3 中的双精度浮点数进行 - rs1*rs2-rs3 操作，将结果写回寄存器 rd 中。
- fnmsub.d 指令将操作数寄存器 rs1、rs2 与 rs3 中的双精度浮点数进行 - rs1*rs2+rs3 操作，将结果写回寄存器 rd 中。

注意：对于上述指令，如果两个被乘数的值为无穷大和 0，则需要在 fscr 寄存器中产生 NV 异常标志。

13. 浮点数格式转换指令

FCVT.W.S，FCVT.S.W，FCVT.WU.S，FCVT.S.WU，FCVT.W.D，FCVT.D.W，FCVT.WU.D，FCVT.D.WU 指令

（1）指令汇编格式

```
fcvt.w.s     rd, rs1
fcvt.s.w     rd, rs1
fcvt.uw.s    rd, rs1
fcvt.s.uw    rd, rs1
fcvt.w.d     rd, rs1
fcvt.d.w     rd, rs1
fcvt.uw.d    rd, rs1
fcvt.d.uw    rd, rs1
```

（2）指令详解

该组指令进行浮点与整数之间的转换操作。

- fcvt.w.s 指令将通用浮点寄存器 rs1 中的单精度浮点数转换成有符号整数，将结果写回通用整数寄存器 rd 中。
- fcvt.s.w 指令将通用整数寄存器 rs1 中的有符号整数转换成为单精度浮点数，将结果写回通用浮点寄存器 rd 中。
- fcvt.uw.s 指令将通用浮点寄存器 rs1 中的单精度浮点数转换成无符号整数，将结果写回通用整数寄存器 rd 中。
- fcvt.s.uw 指令将通用整数寄存器 rs1 中的无符号整数转换成为单精度浮点数，将结果

写回通用浮点寄存器 rd 中。

- fcvt.w.d 指令将通用浮点寄存器 rs1 中的双精度浮点数转换成有符号整数，将结果写回通用整数寄存器 rd 中。
- fcvt.d.w 指令将通用整数寄存器 rs1 中的有符号整数转换成为双精度浮点数，将结果写回通用浮点寄存器 rd 中。
- fcvt.uw.d 指令将通用浮点寄存器 rs1 中的双精度浮点数转换成无符号整数，将结果写回通用整数寄存器 rd 中。
- fcvt.d.uw 指令将通用整数寄存器 rs1 中的无符号整数转换成为双精度浮点数，将结果写回通用浮点寄存器 rd 中。

注意：由于浮点数的表示范围远远大于整数的表示范围，且浮点数存在某些特殊的类型（无穷大或者 NaN），因此将浮点数转换成整数的过程中存在诸多特殊情况，其转换成为整数的结果如图 A-11 所示。

	FCVT.W.S	FCVT.WU.S
Minimum valid input (after rounding)	-2^{31}	0
Maximum valid input (after rounding)	$2^{31}-1$	$2^{32}-1$
Output for out-of-range negative input	-2^{31}	0
Output for $-\infty$	-2^{31}	0
Output for out-of-range positive input	$2^{31}-1$	$2^{32}-1$
Output for $+\infty$ or NaN	$2^{31}-1$	$2^{32}-1$

图 A-11　单精度浮点数转换成整数需处理的特殊情况（双精度同理）

FCVT.S.D，FCVT.D.S 指令

（1）指令汇编格式

```
fcvt.s.d        rd, rs1
fcvt.d.s        rd, rs1
```

（2）指令详解

该组指令进行双精度浮点与单精度浮点之间的转换操作。

- fcvt.s.d 指令将操作数寄存器 rs1 中的双精度浮点数转换成单精度浮点数，将结果写回寄存器 rd 中。
- fcvt.d.s 指令将操作数寄存器 rs1 中的单精度浮点数转换成双精度浮点数，将结果写回寄存器 rd 中。

14．浮点数符号注入指令

FSGNJ，FSGNJN，FSGNJX 指令

（1）指令汇编格式

```
fsgnj.s      rd, rs1, rs2
fsgnjn.s     rd, rs1, rs2
fsgnjx.s     rd, rs1, rs2
fsgnj.d      rd, rs1, rs2
fsgnjn.d     rd, rs1, rs2
fsgnjx.d     rd, rs1, rs2
```

（2）指令详解

该组符号注入指令（Sign-injection Instructions）进行符号注入操作。

- fsgnj.s 指令的操作数均为单精度浮点数，结果的符号位来自操作数寄存器 rs2 的符号

位，结果的其他位来自操作数寄存器 rs1，将结果写回寄存器 rd。

- fsgnjn.s 指令的操作数均为单精度浮点数，结果的符号位来自操作数寄存器 rs2 的符号位取反，结果的其他位来自操作数寄存器 rs1，将结果写回寄存器 rd。
- fsgnjx.s 指令的操作数均为单精度浮点数，结果的符号位来自操作数寄存器 rs1 的符号位与操作数寄存器 rs2 的符号位进行异或（xor）操作，结果的其他位来自操作数寄存器 rs1，将结果写回寄存器 rd。
- fsgnj.d 指令的操作数均为双精度浮点数，结果的符号位来自操作数寄存器 rs2 的符号位，结果的其他位来自操作数寄存器 rs1，将结果写回寄存器 rd。
- fsgnjn.d 指令的操作数均为双精度浮点数，结果的符号位来自操作数寄存器 rs2 的符号位取反，结果的其他位来自操作数寄存器 rs1，将结果写回寄存器 rd。
- fsgnjx.d 指令的操作数均为双精度浮点数，结果的符号位来自操作数寄存器 rs1 的符号位与操作数寄存器 rs2 的符号位进行异或（xor）操作，结果的其他位来自操作数寄存器 rs1，将结果写回寄存器 rd。

注意：

- 使用上述指令的不同形式可以等效为不同的伪指令，譬如 FMV、FNEG 和 FABS 等。请参见附录 G 了解更多伪指令信息。
- FSGNJ、FSGNJN 和 FSGNJX 指令对于 NaN 类型的操作数并不做特殊对待，而是将其当作普通操作数一样进行符号注入操作。

15．浮点与整数互搬指令

FMV.X.W，FMV.W.X 指令

（1）指令汇编格式

```
fmv.x.w        rd, rs1
fmv.w.x        rd, rs1
```

（2）指令详解

该组指令进行浮点与整数寄存器之间的数据搬运操作。

- fmv.x.w 指令将通用浮点寄存器 rs1 中的单精度浮点数读出，然后写回通用整数寄存器 rd 中。
- fmv.w.x 指令将通用整数寄存器 rs1 中的整数读出，然后写回通用浮点寄存器 rd 中。

注意：由于 32 位架构的通用整数寄存器的宽度为 32 位，而双精度浮点数为 64 位，无法实现双精度浮点寄存器与整数寄存器之间的数据互相搬运，因此在 32 位架构中没有此类指令。

16．浮点数比较指令

FLT，FLE，FEQ 指令

（1）指令汇编格式

```
flt.s    rd, rs1, rs2
fle.s    rd, rs1, rs2
feq.s    rd, rs1, rs2
flt.d    rd, rs1, rs2
fle.d    rd, rs1, rs2
feq.d    rd, rs1, rs2
```

（2）指令详解

该组指令进行浮点数的比较操作。

- flt.s 指令：如果通用浮点寄存器 rs1 中的单精度浮点数值小于 rs2 中的值，则结果为 1，否则为 0，将结果写回通用整数寄存器 rd 中。
- fle.s 指令：如果通用浮点寄存器 rs1 中的单精度浮点数值小于或者等于 rs2 中的值，则结果为 1，否则为 0，将结果写回通用整数寄存器 rd 中。
- feq.s 指令：如果通用浮点寄存器 rs1 中的单精度浮点数值等于 rs2 中的值，则结果为 1，否则为 0，将结果写回通用整数寄存器 rd 中。
- flt.d 指令：如果通用浮点寄存器 rs1 中的双精度浮点数值小于 rs2 中的值，则结果为 1，否则为 0，将结果写回通用整数寄存器 rd 中。
- fle.d 指令：如果通用浮点寄存器 rs1 中的双精度浮点数值小于或者等于 rs2 中的值，则结果为 1，否则为 0，将结果写回通用整数寄存器 rd 中。
- feq.d 指令：如果通用浮点寄存器 rs1 中的双精度浮点数值等于 rs2 中的值，则结果为 1，否则为 0，将结果写回通用整数寄存器 rd 中。

注意：

- 对于 FLT、FLE 和 FEQ 指令，如果任何一个操作数为 NaN，则结果为 0。
- 对于 FLT 和 FLE 指令，如果任意一个操作数属于 NaN，则需要在 fscr 寄存器中产生 NV 异常标志。
- 对于 FEQ 指令，如果任意一个操作数属于 Signaling-NaN，则需要在 fscr 寄存器中产生 NV 异常标志。

17. 浮点数分类指令

FCLASS 指令

（1）指令汇编格式

```
fclass.s        rd, rs1
fclass.d        rd, rs1
```

（2）指令详解

该组指令进行浮点数的分类操作。

- fclass.s 指令：对通用浮点寄存器 rs1 中的单精度浮点数进行判断，根据其所属的类型，生成一个 10 位的独热码（one-hot）结果，结果的每一位对应一种类型，如图 A-12 所

示，将结果写回通用整数寄存器 rd 中。

- fclass.d 指令：对通用浮点寄存器 rs1 中的双精度浮点数进行判断，根据其所属的类型，生成一个 10 位的独热码（one-hot）结果，结果的每一位对应一种类型，如图 A-12 所示，将结果写回通用整数寄存器 rd 中。

rd bit	Meaning
0	*rs1* is $-\infty$.
1	*rs1* is a negative normal number.
2	*rs1* is a negative subnormal number.
3	*rs1* is -0.
4	*rs1* is $+0$.
5	*rs1* is a positive subnormal number.
6	*rs1* is a positive normal number.
7	*rs1* is $+\infty$.
8	*rs1* is a signaling NaN.
9	*rs1* is a quiet NaN.

图 A-12　浮点分类指令的分类结果

A14.5　存储器原子操作指令（RV32A 指令子集）

本节介绍 RISC-V 架构的原子操作指令，在 RISC-V 的“指令集文档”中并未对存储器原子操作指令进行系统解释，因为“指令集文档”是对 RISC-V 架构的精确定义，而非计算机体系结构的教学文章。为了便于读者理解，本书单独设立附录 E 对原子操作指令的相关知识背景予以简介，建议读者先参见附录 E 中的相关背景介绍。

阅读了附录 E 的读者，想必已了解 RISC-V 架构定义了可选的（非必需的）存储器原子操作指令（A 扩展指令子集）。该扩展指令子集支持两类指令：

- Atomic-Memory-Operation（AMO）指令
- Load-Reserved 和 Store-Conditional 指令

1. Atomic-Memory-Operation（AMO）指令

AMO 指令

注意：本节仅介绍 RISC-V 32 位架构的 AMO 指令。

（1）指令汇编格式

```
amoswap.w   rd, rs2, (rs1)
amoadd.w    rd, rs2, (rs1)
amoand.w    rd, rs2, (rs1)
amoor.w     rd, rs2, (rs1)
amoxor.w    rd, rs2, (rs1)
amomax.w    rd, rs2, (rs1)
amomaxu.w   rd, rs2, (rs1)
amomin.w    rd, rs2, (rs1)
amominu.w   rd, rs2, (rs1)
```

（2）指令详解

此类指令用于从存储器（地址为 rs1 寄存器的值指定）中读出一个数据，存放至 rd 寄存器中，并且将读出的数据与 rs2 寄存器的值进行计算，再将计算后的结果写回存储器（存储器写回地址与读出地址相同）。

对读出数据进行的计算操作类型依赖于具体的指令类型。

- amoswap.w　将读出的数据与 rs2 寄存器的值进行互换。
- amoadd.w　将读出的数据与 rs2 寄存器的值进行加法操作。
- amoand.w　将读出的数据与 rs2 寄存器的值进行与操作。
- amoor.w　将读出的数据与 rs2 寄存器的值进行或操作。
- amoxor.w　将读出的数据与 rs2 寄存器的值进行异或操作。
- amomax.w　将读出的数据与 rs2 寄存器的值进行（当作有符号数）取最大值操作。
- amomaxu.w　将读出的数据与 rs2 寄存器的值进行（当作无符号数）取最大值操作。
- amomin.w　将读出的数据与 rs2 寄存器的值进行（当作有符号数）取最小值操作。
- amominu.w　将读出的数据与 rs2 寄存器的值进行（当作无符号数）取最小值操作。

对于 32 位架构的 AMO 指令，访问存储器的地址必须与 32 位对齐，否则会产生地址非对齐异常（AMO Misaligned Address Exception）。

AMO 指令要求整个"读出-计算-写回"过程必须为"Atomic（原子）"性质。所谓原子性质即整个"读出-计算-写回"过程必须能够确切完成，在读出和写回之间的间隙，存储器的该地址不能够被其他的进程访问（通常会将总线锁定）。请参见附录 E 了解更多背景知识和相关信息。

AMO 指令还支持释放一致性模型（Release Consistency Model），有关释放一致性模型的知识背景请参见附录 D2.3 节。如图 A-13 所示，AMO 指令的编码中包含了 aq/rl 编码位，分别表示获取或者释放操作。通过设置不同的编码位，就可以赋予 AMO 指令获取或者释放操作属性，有关获取或者释放操作属性的含义，请参见附录 D2.3 节。

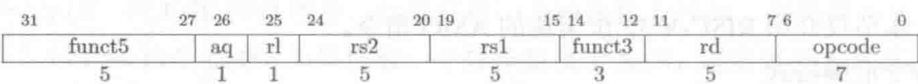


图 A-13　AMO 指令的指令编码

- amoswap.w　rd, rs2, (rs1)指令不具有获取和释放属性，不具备屏障功能。
- amoswap.w.aq　rd, rs2, (rs1)指令具有获取属性，能够屏障其之后的所有存储器访问操作。
- amoswap.w.rl　rd, rs2, (rs1) 指令具有释放属性，能够屏障其之前的所有存储器访问操作。
- amoswap.w.aqrl　rd, rs2, (rs1) 指令同时具有获取和释放属性，能够屏障其之前和之后的所有存储器访问操作。

使用带有获取或者释放属性的 AMO 指令可以实现如附录 E 中介绍的"上锁"操作。其示例程序代码如下：

```
li t0, 1                    # 将 T0 寄存器的值初始化为 1
again:
amoswap.w.aq t0, t0, (a0) # 使用带获取属性的 amoswap 指令，将存在（a0）地址中的
```

```
                           #    锁的值读出，并将 t0 之前的值写入（a0）地址。
bnez t0, again             # 如果锁中的值非 0，意味着当前的锁仍然被其他进程占用，因此重
                           #    新读取锁的值。
# ...                      # 否则，如果锁中的值为 0，则意味着上锁成功，可以进行独占后的
                           #    后续操作。
# Critical section.
# ...
amoswap.w.rl x0, x0, (a0)  # 完成操作后，通过带有释放属性的 amoswap 指令向锁中写
                           #    入数值 0，将锁释放。
```

2．Load-Reserved 和 Store-Conditional 指令

Load-Reserved/Store-Conditional 指令

注意：本节仅介绍 RISC-V 32 位架构的 Load-Reserved 和 Store-Conditional 指令。

（1）指令汇编格式

```
lr.w    rd, (rs1)
sc.w    rd, rs2, (rs1)
```

（2）指令详解

Load-Reserved 和 Store-Conditional 指令的功能与附录 E3 中介绍的互斥读（Load-Exclusive）和互斥写（Store-Exclusive）指令完全相同，请参见附录 E 了解更多相关背景知识。

LR（Load-Reserved）指令用于从存储器（地址为 rs1 寄存器的值指定）中读出一个 32 位数据，存放至 rd 寄存器中。

SC（Store-Conditional）指令用于向存储器（地址为 rs1 寄存器的值指定）中写入一个 32 位数据，数据的值来自于 rs2 寄存器中的值。SC 指令不一定能够执行成功，只有满足如下条件，SC 指令才能够执行成功。

- LR 和 SC 指令成对地访问相同的地址。
- LR 和 SC 指令之间没有任何其他的写操作（来自任何一个 Hart）访问过同样的地址。
- LR 和 SC 指令之间没有任何中断与异常发生。
- LR 和 SC 指令之间没有执行 MRET 指令。

如果执行成功，则向 rd 寄存器写回数值 0，如果执行失败，则向 rd 寄存器写回一个非零值；如果执行失败，意味着没有真正写入存储器。

对于 32 位架构的 LR 和 SC 指令，访问存储器的地址必须与 32 位对齐，否则会产生地址非对齐异常（misaligned address exception）。

LR/SC 指令也支持释放一致性模型，有关释放一致性模型的知识背景请参见附录 D2.3 节。如图 A-14 所示，LR/SC 指令的编码中包含了 aq/rl 编码位，分别表示获取（acquire）或者释放（release）操作。与 AMO 指令相同，通过设置不同的编码位，就可以赋予 LR/SC 指令获取或者释放操作属性，有关获取或者释放操作属性的含义，请参见附录 D2.3 节。

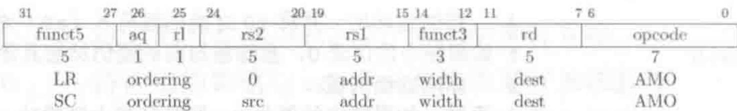


图 A-14 LR/SC 指令的指令编码

A14.6 16 位压缩指令（RV32C 指令子集）

本节介绍 RISC-V 架构的 16 位长度编码的压缩指令（C 扩展指令子集）。有关 RISC-V 架构压缩指令的特点和概述，请参见第 2.2.11 节。

如第 2.2.11 节所述， RISC-V 架构的精妙之处在于每一条 16 位的指令都能找到一一对应的原始 32 位指令。本节将 RISC-V 32 位架构下的压缩指令（RV32C）进行列举，并给出其对应的原始 32 位指令，如表 A-1 所示。对于每条 16 位指令的具体描述，本节不再赘述，请参见其相应的 32 位指令功能描述，或 RISC-V 架构“指令集文档”原文。

注意：由于 16 位指令的编码长度有限，因此有的指令只能使用 8 个最为常用的通用寄存器作为操作数，即编号 x8～x15 的 8 个通用寄存器（如果使用的是浮点通用寄存器，则为 f8～f15），但有的指令还是可以使用所有的通用寄存器作为操作数。有关 RV32C 指令的详细编码请参见附录 F6。

表 A-1 RV32C 指令列表

注意：

- 表格中仅介绍 RISC-V 32 位架构的压缩指令（RV32C）
- 某些压缩指令的操作数寄存器索引不能为特定值，譬如 rs1 索引不能等于 0，否则为非法指令。有关每条指令的具体非法情形，请参见附录 F6 中 RVC 指令编码图表

指令分组	16 位指令	对应 32 位指令	注意事项
Stack-Pointer-Based Loads and Stores	c.lwsp rd, offset[7:2]	lw rd, offset[7:2](x2)	• 此指令可以使用所有的通用寄存器作为操作数
	c.flwsp rd, offset[7:2]	flw rd, offset[7:2](x2)	
	c.fldsp rd, offset[8:3]	fld rd, offset[8:3](x2)	
	c.swsp rs2,offset[7:2]	sw rs2,offset[7:2](x2)	
	c.fswsp rs2, offset[7:2]	fsw rs2, offset[7:2](x2)	
	c.fsdsp rs2, offset[8:3]	fsd rs2, offset[8:3](x2)	
Register-Based Loads and Stores	c.lw rd, offset[6:2](rs1)	lw rd, offset[6:2](rs1)	• 此指令只能够使用 8 个最为常用的通用寄存器作为操作数（其中，c.flw/c.fld 的 rd 和 c.fsw/c.fsd 的 rs2 为通用浮点寄存器）
	c.flw rd, ffset[6:2](rs1)	flw rd, ffset[6:2](rs1)	
	c.fld rd,offset[7:3](rs1)	fld rd,offset[7:3](rs1)	
	c.sw rs2,offset[6:2](rs1)	sw rs2,offset[6:2](rs1)	
	c.fsw rs2,offset[6:2](rs1)	fsw rs2,offset[6:2](rs1)	
	c.fsd rs2,offset[7:3](rs1)	fsd rs2,offset[7:3](rs1)	
Control Transfer Instructions	c.j offset[11:1]	jal x0,offset[11:1]	—
	c.jal offset[11:1]	jal x1, offset[11:1]	—
	c.jr rs1	jalr x0, rs1, 0	• 此指令可以使用所有的通用寄存器作为操作数
	c.jalr rs1	jalr x1, rs1, 0	

续表

<table>
<tr><th>指令分组</th><th>16 位指令</th><th>对应 32 位指令</th><th>注意事项</th></tr>
<tr><td rowspan="2">Control Transfer Instructions</td><td>c.beqz rs1 offset[8:1]</td><td>beq rs1, x0, offset[8:1]</td><td rowspan="2">• 此指令只能够使用 8 个最为常用的通用寄存器作为操作数</td></tr>
<tr><td>c.bnez rs1 offset[8:1]</td><td>bne rs1, x0, offset[8:1]</td></tr>
<tr><td rowspan="16">Integer Computational Instructions</td><td>c.li rd, imm[5:0]</td><td>addi rd, x0, imm[5:0]</td><td rowspan="3">• 此指令可以使用所有的通用寄存器作为操作数</td></tr>
<tr><td>c.lui rd, nzuimm[17:12]</td><td>lui rd, nzuimm[17:12]</td></tr>
<tr><td>c.addi rd, nzimm[5:0]</td><td>addi rd, rd, nzimm[5:0]</td></tr>
<tr><td>c.addi16sp nzimm[9:4]</td><td>addi x2, x2, nzimm[9:4]</td><td>—</td></tr>
<tr><td>c.addi4spn rd, nzuimm[9:2]</td><td>addi rd, x2, nzuimm[9:2]</td><td>• 此指令只能够使用 8 个最为常用的通用寄存器作为操作数</td></tr>
<tr><td>c.slli rd, shamt[5:0]</td><td>slli rd, rd, shamt[5:0]</td><td>• 此指令可以使用所有的通用寄存器作为操作数</td></tr>
<tr><td>c.srli rd, rd, shamt[5:0]</td><td>srli rd, rd, shamt[5:0]</td><td rowspan="3">• 此指令只能够使用 8 个最为常用的通用寄存器作为操作数</td></tr>
<tr><td>c.srai rd, shamt[5:0]</td><td>srai rd, rd, shamt[5:0]</td></tr>
<tr><td>c.andi rd, imm[5:0]</td><td>andi rd, rd, imm[5:0]</td></tr>
<tr><td>c.mv rd, rs2</td><td>add rd, x0, rs2</td><td rowspan="2">• 此指令可以使用所有的通用寄存器作为操作数</td></tr>
<tr><td>c.add rd, rs2</td><td>add rd, rd, rs2</td></tr>
<tr><td>c.and rd, rs2</td><td>and rd, rd, rs2</td><td rowspan="4">• 此指令只能够使用 8 个最为常用的通用寄存器作为操作数</td></tr>
<tr><td>c.or rd, rs2</td><td>or rd, rd, rs2</td></tr>
<tr><td>c.xor rd, rs2</td><td>xor rd, rd, rs2</td></tr>
<tr><td>c.sub rd, rs2</td><td>sub rd, rd, rs2</td></tr>
<tr><td>NOP Instruction</td><td>c.nop</td><td>addi x0, x0, 0.</td><td>• 32 位的 nop 指令对应的实际指令编码也是 addi x0, x0, 0</td></tr>
<tr><td>Breakpoint Instruction</td><td>c.ebreak</td><td>ebreak</td><td>—</td></tr>
<tr><td>Defined Illegal Instruction</td><td colspan="3">RISC-V 架构规定，对于任意长度编码的指令，只要编码是全 0 或者全 1 都是非法指令，这个特性在抓出某些特殊错误情况非常有用，譬如取指进入全 0 的数据段，未连接的总线或者未初始化的存储器段等</td></tr>
</table>

A15 伪指令

RISC-V 架构定义了一系列的伪指令，所谓伪指令意味着它并不是一条真正的指令，而是对其他基本指令使用形式的一种别名，请参见附录 G 了解完整的伪指令列表。

A16 指令编码

RV32GC 的完整指令列表及其编码，请参见附录 F。

附录 B RISC-V 架构 CSR 寄存器介绍

RISC-V 的架构中定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用其专有的 12 位地址编码空间。

附录对于 CSR 寄存器的介绍翻译自 RISC-V 的“特权架构文档”，本书对相关内容进行了重新组织，以求通俗易懂。

注意：附录仅介绍 RV32GC，且只支持机器模式（Machine Mode Only）相关的 CSR 寄存器。有关 RISC-V 所有 CSR 寄存器的完整介绍，感兴趣的读者请参见 RISC-V “特权架构文档”原文。

B1 蜂鸟 E200 支持的 CSR 寄存器列表

蜂鸟 E200 支持的 CSR 寄存器列表如表 B-1 所示，其中包括 RISC-V 标准的 CSR 寄存器（RV32GC 且只支持机器模式相关）和蜂鸟 E200 自定义扩展的 CSR 寄存器。

注意：有关每个寄存器的详细解释，请参见 B2。

表 B-1 蜂鸟 E200 支持的 CSR 寄存器列表

类型	CSR 地址	读写属性	名 称	全 称
RISC-V 标准 CSR	0x001	MRW	fflags	浮点累积异常（Floating-Point Accrued Exceptions）
	0x002	MRW	frm	浮点动态舍入模式（Floating-Point Dynamic Rounding Mode）
	0x003	MRW	fcsr	浮点控制和状态寄存器（Floating-Point Control and Status Register ）
	0x300	MRW	mstatus	机器模式状态寄存器（Machine Status Register）
	0x301	MRW	misa	机器模式指令集架构寄存器（Machine ISA Register）
	0x304	MRW	mie	机器模式中断使能寄存器（Machine Interrupt Enable Registers）
	0x305	MRW	mtvec	机器模式异常入口基地址寄存器（Machine Trap-Vector Base-Address Register）
	0x340	MRW	mscratch	机器模式擦写寄存器（Machine Scratch Register）
	0x341	MRW	mepc	机器模式异常 PC 寄存器（Machine Exception Program Counter）
	0x342	MRW	mcause	机器模式异常原因寄存器（Machine Cause Register）

续表

类型	CSR 地址	读写属性	名 称	全 称
RISC-V 标准 CSR	0x343	MRW	mtval (又名 mbadaddr)	机器模式异常值寄存器（Machine Trap Value Register）
	0x344	MRW	mip	机器模式中断等待寄存器（Machine Interrupt Pending Registers）
	0xB00	MRW	mcycle	周期计数器的低 32 位（Lower 32 bits of Cycle counter）
	0xB80	MRW	mcycleh	周期计数器的高 32 位（Upper 32 bits of Cycle counter）
	0xB02	MRW	minstret	退休指令计数器的低 32 位（Lower 32 bits of Instructions-retired counter）
	0xB82	MRW	minstreth	退休指令计数器的高 32 位（Upper 32 bits of Instructions-retired counter）
	0xF11	MRW	mvendorid	机器模式供应商编号寄存器（Machine Vendor ID Register）
	0xF12	MRO	marchid	机器模式架构编号寄存器（Machine Architecture ID Register）
	0xF13	MRO	mimpid	机器模式硬件实现编号寄存器（Machine Implementation ID Register）
	0xF14	MRO	mhartid	Hart 编号寄存器（Hart ID Register）
	N/A	MRW	mtime	机器模式计时器寄存器（Machine-mode timer register）
	N/A	MRW	mtimecmp	机器模式计时器比较寄存器（Machine-mode timer compare register）
	N/A	MRW	msip	机器模式软件中断等待寄存器（Machine-mode Software Interrupt Pending Register）
蜂鸟 E200 自定义CSR	0xBFF	MRW	mcounterstop	自定义寄存器用于停止 mtime、mcycle、mcycleh、minstret 和 minstreth 对应的计数器

B2 RISC-V 标准 CSR

本节介绍 RISC-V 架构 RV32GC，且只支持机器模式相关的 CSR 寄存器。

B2.1 misa

misa 寄存器用于指示当前处理器所支持的架构特性。

misa 寄存器的最高两位用于指示当前处理器所支持的架构位数。

- 如果最高两位值为 1，则表示当前为 32 位架构（RV32）。
- 如果最高两位值为 2，则表示当前为 64 位架构（RV64）。
- 如果最高两位值为 3，则表示当前为 128 位架构（RV128）。

misa 寄存器的低 26 位用于指示当前处理器所支持的 RISC-V ISA 中不同模块化指令子集，每一位表示的模块化指令子集如图 B-1 所示。

注意：misa 寄存器在 RISC-V 架构文档中被定义为可读可写的寄存器，从而允许某些处理器的设计能够动态地配置某些特性。但是在蜂鸟 E200 的实现中，misa 寄存器为只读寄存

器，恒定地反映不同型号处理器核所支持的 ISA 模块化子集。譬如蜂鸟 E203 核支持 RV32IMAC，则反映于此寄存器中，最高两位值为 1，低 26 位中 I/M/A/C 对应域的值即为高。

Bit	Character	Description
0	A	Atomic extension
1	B	*Tentatively reserved for Bit operations extension*
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E base ISA
5	F	Single-precision floating-point extension
6	G	Additional standard extensions present
7	H	*Reserved*
8	I	RV32I/64I/128I base ISA
9	J	*Tentatively reserved for Dynamically Translated Languages extension*
10	K	*Reserved*
11	L	*Tentatively reserved for Decimal Floating-Point extension*
12	M	Integer Multiply/Divide extension
13	N	User-level interrupts supported
14	O	*Reserved*
15	P	*Tentatively reserved for Packed-SIMD extension*
16	Q	Quad-precision floating-point extension
17	R	*Reserved*
18	S	Supervisor mode implemented
19	T	*Tentatively reserved for Transactional Memory extension*
20	U	User mode implemented
21	V	*Tentatively reserved for Vector extension*
22	W	*Reserved*
23	X	Non-standard extensions present
24	Y	*Reserved*
25	Z	*Reserved*

图 B-1 misa 寄存器低 26 位各域表示的模块化指令子集

B2.2 mvendorid

此寄存器是只读寄存器，用于反映该处理器核的商业供应商编号（Vendor ID）。

如果此寄存器的值为 0，则表示此寄存器未实现，或者表示此处理器不是一个商业处理器核。

B2.3 marchid

此寄存器是只读寄存器，用于反映该处理器核的硬件实现微架构编号（Microarchitecture ID）。

如果此寄存器的值为 0，则表示此寄存器未实现。

B2.4 mimpid

此寄存器是只读寄存器，用于反映该处理器核的硬件实现编号（Implementation ID）。

如果此寄存器的值为 0，则表示此寄存器未实现。

B2.5 mhartid

此寄存器是只读寄存器，用于反映当前 Hart 的编号（Hart ID）。有关 Hart 的概念请参见

附录 A9 中所述。

RISC-V 架构规定，如果在单 Hart 或者多 Hart 的系统中，起码要有一个 Hart 的编号必须是 0。

B2.6 fflags

fflags 寄存器为浮点控制状态寄存器（fcsr）中浮点异常标志位（Accrued Exceptions）域的别名。之所以单独定义一个 fflags 寄存器，是为了方便使用 CSR 指令直接地单独读写浮点异常标志位域。

请参见附录 A14.4 节了解更多浮点异常标志位的相关信息。

B2.7 frm

frm 寄存器为浮点控制状态寄存器中浮点舍入模式（Rounding Mode）域的别名。之所以单独定义一个 frm 寄存器，是为了方便使用 CSR 指令直接地单独读写浮点舍入模式。

请参见附录 A14.4 节了解更多浮点舍入模式的相关信息。

B2.8 fcsr

RISC-V 架构规定，如果支持单精度浮点指令或者双精度浮点指令，则需要增加一个浮点控制状态寄存器。该寄存器包含了浮点异常标志位域（Accrued Exceptions）和浮点舍入模式（Rounding Mode）域。

请参见附录 A14.4 节了解更多浮点控制状态寄存器的相关信息。

B2.9 mstatus

mstatus 寄存器是机器模式（Machine Mode）下的状态寄存器。

如图 B-2 所示，该寄存器包含若干不同的功能域，其中 TSR、TW、TVM、MXR、SUM、MPRV、SPP、SPIE、UPIE、SIE 以及 UIE 域与本书介绍的配置（RV32GC 且只支持机器模式）无关，因此在此不做介绍。本书仅对剩余的 SD、XS、FS、MPP、MPIE 以及 MIE 域予以介绍。

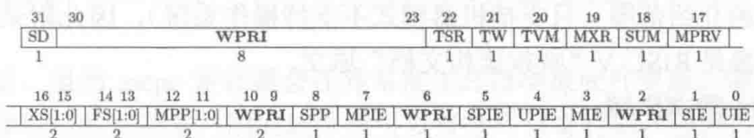


图 B-2 mstatus 寄存器格式

1. mstatus 的 MIE 域

mstatus 寄存器中的 MIE 域表示全局中断使能。当该 MIE 域的值为 1 时，表示所有中断的全局开关打开；当 MIE 域的值为 0 时，表示全局关闭所有的中断。

为了理解此寄存器，请先参见第 13 章系统地了解中断和异常的相关信息。

2. mstatus 的 MPIE、MPP 域

mstatus 寄存器中的 MPIE 和 MPP 域分别用于保存进入异常之前 MIE 域和特权模式（Privilege Mode）的值。

为了理解此寄存器，请先参见第 13 章系统地了解中断和异常的相关信息。

RISC-V 架构规定，处理器进入异常时：

- MPIE 域的值被更新为当前 MIE 的值。
- MIE 的值则被更新成为 0（意味着进入异常服务程序后中断被屏蔽）。
- MPP 的值被更新为异常发生前的模式（如果是只支持机器模式，则 MPP 的值永远为 11）。

3. mstatus 的 FS 域

mstatus 寄存器中的 FS 域用于维护或反映浮点单元的状态，FS 域由两位组成，其编码如图 B-3 所示。

Status	FS Meaning	XS Meaning
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

图 B-3 FS 域表示的状态编码

FS 域的更新准则如下。

- FS 上电后的默认值为 0，意味着浮点单元的状态为 Off。因此为了能够正常使用浮点单元，软件需要使用 CSR 写指令将 FS 的值改写为非 0 值，以打开浮点单元的功能。
- 如果 FS 的值为 1 或者 2，当执行了任何的浮点指令之后，FS 的值会自动切换为 3，表示浮点单元的状态为脏（Dirty）（状态发生了改变）。
- 如果处理器不想使用浮点运算单元（譬如将浮点单元关电以节省功耗），可以使用 CSR 写指令将 mstatus 寄存器的 FS 域设置成 0，将浮点单元的功能予以关闭。当浮点单元的功能关闭之后，任何访问浮点 CSR 寄存器的操作或者任何执行浮点指令的行为都将会产生非法指令（Illegal Instruction）异常。

除了用于上述功能，FS 域的值还用于操作系统在进行上下文切换时的指引信息，由于此内容超出本书的介绍范围（只支持机器模式不支持操作系统），因此附录在此不做介绍。感兴趣的读者请参见 RISC-V“特权架构文档”原文。

4. mstatus 的 XS 域

mstatus 寄存器中的 XS 域与 FS 域的作用类似，但是其用于维护或反映用户自定义的扩展指令单元状态。

在标准的 RISC-V“特权架构文档”中定义 XS 域为只读域，其用于反映所有自定义扩展指令单元的状态总和。但请注意：在蜂鸟 E200 的硬件实现中，将 XS 域设计成可写可读域，其作用完全与 FS 域类似，软件可以通过改写 XS 域的值达到打开或者关闭协处理器扩展指令单元的目的。请参见第 16 章了解更多蜂鸟 E200 协处理器扩展的相关信息。

与 FS 域类似，XS 除了用于上述功能之外，还用于操作系统在进行上下文切换时的指引信息。由于此内容超出本书的介绍范围（只支持机器模式不支持操作系统），因此附录在此不做介绍，感兴趣的读者请参见 RISC-V“特权架构文档”原文。

5. mstatus 的 SD 域

mstatus 寄存器中的 SD 域是一个只读域，其反映了 XS 域或者 FS 域处于脏（Dirty）状态。其逻辑关系表达式为：SD=((FS==11) OR (XS==11))。

之所以设置此只读的 SD 域，是为了方便软件快速的查询 XS 域或者 FS 域是否处于脏（Dirty）状态，从而在上下文切换时可以快速判断是否需要对浮点单元或者扩展指令单元进行上下文的保存。由于此内容超出本书的介绍范围（只支持机器模式不支持操作系统），因此在此不做过多介绍，感兴趣的读者请参见 RISC-V“特权架构文档”原文。

B2.10 mtvec

mtvec 寄存器用于配置异常的入口地址。

为了理解此寄存器，请先参见第 13 章系统地了解中断和异常的相关信息。

在处理器的程序执行过程中，一旦遇到异常发生，则终止当前的程序流，处理器被强行跳转到一个新的 PC 地址，该过程在 RISC-V 的架构中定义为陷阱（trap），字面的含义为“跳入陷阱”，更加准确的含义为“进入异常”。RISC-V 处理器进入异常后跳入的 PC 地址即由 mtvec 寄存器指定。

有关 RISC-V 架构定义的 mtvec 寄存器详细格式，请参见第 13.2.1 节。

B2.11 mepc

mepc 寄存器用于保存进入异常之前指令的 PC 值，作为异常的返回地址。

为了理解此寄存器，请先参见第 13 章，系统地了解中断和异常的相关信息。

RISC-V 架构规定，处理器进入异常时，mepc 寄存器被同时更新以反映当前遇到异常的指令的 PC 值。

值得注意的是，虽然 mepc 寄存器会在异常发生时自动被硬件更新，但是 mepc 寄存器本身也是一个可读可写的寄存器，因此软件也可以直接写该寄存器，以修改它的值。

注意：RISC-V 在中断和异常时的返回地址定义（更新 mepc 的值）有如下细微差别。

- 出现中断时，中断返回地址 mepc 被指向下一条尚未执行的指令，因为中断时的指令

被正确执行。

- 出现异常时，mepc 则指向当前指令，因为当前指令触发了异常。

如第 13.1.3 节中所述，同步异常能够精确定位到造成异常发生的指令，而对于异步异常，则无法精确定位，这取决于处理器的具体硬件实现。

如果异常由 ecall 或 ebreak 产生，直接跳回返回地址则会造成死循环(因为重新执行 ecall 导致重新进入异常)。正确的做法是在异常处理中软件改变 mepc 指向下一条指令，由于现在 ecall/ebreak 都是 4 字节指令，因此简单设定 mepc=mepc+4 即可。

B2.12 mcause

mcause 寄存器，用于保存进入异常之前的出错原因，以便对异常原因进行诊断和调试。

为了理解此寄存器，请先参见第 13 章系统地了解中断和异常的相关信息。

RISC-V 架构规定，处理器进入异常时，mcause 寄存器被同时更新以反映当前遇到异常的原因：mcause 寄存器的最高 1 位为中断(Interrupt)域，低 31 位为异常编号(Exception Code)域，此两个域的组合可以用于指示 12 种定义的中断类型和 16 种定义的异常类型。

有关 RISC-V 架构定义的 mcause 寄存器详细格式，以及蜂鸟 E200 支持的中断和异常原因类型，请参见第 13.2.1 节。

B2.13 mtval (mbadaddr)

mtval（又名 mbadaddr）寄存器，用于保存进入异常之前的出错指令的编码值或者存储器访问的地址值，以便对异常原因进行诊断和调试。

为了理解此寄存器，请先参见第 13 章，系统地了解中断和异常的相关信息。

RISC-V 架构规定，处理器进入异常时，mtval 寄存器被同时更新以反映当前遇到异常的信息。

- 如果是与存储器访问造成的异常，譬如硬件断点、取指令和存储器读写造成的异常，则将存储器访问的地址值更新到 mtval 寄存器中。
- 如果是非法指令造成的异常，则将错误的指令编码更新到 mtval 寄存器中。

B2.14 mie

mie 寄存器用于控制不同中断类型的局部屏蔽。之所以称为局部屏蔽，是因为相对而言 mstatus 寄存器中的 MIE 域提供了全局中断使能，请参见附录 B2.9 节对 mstatus 寄存器了解更多信息。

为了理解此寄存器，请先参见第 13 章，系统地了解中断和异常的相关信息。

RISC-V 架构对于 mie 寄存器的规定如下。

- mie 寄存器的每一个域用于控制每个单独的中断使能，MEIE/MTIE/MSIE 域分别控制机器模式下的外部中断（External Interrupt）、计时器中断（Timer Interrupt）和软件中断（Software Interrupt）的屏蔽。如果处理器（譬如蜂鸟 E200）只实现了机器模式，则监督模式（Supervisor）和用户模式（User Mode）对应的中断使能位（SEIE、UEIE、STIE、UTIE、SSIE 以及 USIE）无任何意义。
- 有关 RISC-V 架构定义的 mie 寄存器详细格式和功能，以及蜂鸟 E200 支持的中断类型，请参见第 13.3.2 节。

B2.15 mip

mip 寄存器用于查询中断的等待（Pending）状态。

为了理解此寄存器，请先参见第 13 章。系统了解中断和异常的相关信息。

RISC-V 架构对于 mip 寄存器的规定如下。

- mip 寄存器的中的每一个域用于反映每个单独的中断等待状态，MEIP/MTIP/MSIP 域分别反映机器模式下的外部中断、计时器中断和软件中断的等待状态。如果处理器（譬如蜂鸟 E200）只实现了机器模式，则监督模式和用户模式对应的中断等待状态位（SEIP、UEIP、STIP、UTIP、SSIP 以及 USIP）无任何意义。
- 有关 RISC-V 架构定义的 mip 寄存器详细格式和功能，以及蜂鸟 E200 支持的中断类型，请参见第 13.3.3 节。

B2.16 mscratch

mscratch 寄存器用于机器模式下的程序临时保存某些数据。mscratch 寄存器可以提供一种快速的保存和恢复机制。譬如，在进入机器模式的异常处理程序后，将应用程序的某个通用寄存器的值临时存入 mscratch 寄存器中，然后在退出异常处理程序之前，将 mscratch 寄存器中的值读出恢复至通用寄存器。

B2.17 mcycle 和 mcycleh

RISC-V 架构定义了一个 64 位宽的时钟周期计数器，用于反映处理器执行了多少个时钟周期。只要处理器处于执行状态，此计数器便会不断自增计数，其自增的时钟频率由处理器的硬件实现自定义。

mcycle 寄存器反映了该计数器低 32 位的值，mcycleh 寄存器反映了该计数器高 32 位的值。

mcycle 和 mcycleh 寄存器可以用于衡量处理器的性能，且具备可读可写属性，因此软件

可以通过 CSR 指令改写 mcycle 和 mcycleh 寄存器中的值。

考虑到此计数器计数会消耗某些动态功耗，在蜂鸟 E200 处理器的实现中，在自定义 mcounterstop 寄存器中额外增加了一位控制域。软件可以配置此控制域将 mcycle 和 mcycleh 对应的计数器停止计数，从而在不需要衡量性能之时停止计数器，以达到省电的作用。请参见附录 B3.1 节，了解更多 mcounterstop 寄存器信息。

B2.18 minstret 和 minstreth

RISC-V 架构定义了一个 64 位宽的执行指令计数器，用于反映处理器成功执行了多少条指令。只要处理器每成功执行一条指令，此计数器便会自增计数。

minstret 寄存器反映了该计数器低 32 位的值，minstreth 寄存器反映了该计数器高 32 位的值。

minstret 和 minstreth 寄存器可以用于衡量处理器的性能，且具备可读可写属性，因此软件可以通过 CSR 指令改写 minstret 和 minstreth 寄存器中的值。

考虑到此计数器计数会消耗某些动态功耗，在蜂鸟 E200 处理器的实现中，在自定义 mcounterstop 寄存器中额外增加了一位控制域。软件可以配置此控制域将 minstret 和 minstreth 对应的计数器停止计数，从而在不需要衡量性能之时停止计数器，以达到省电的作用。请参见附录 B3.1 节，了解更多 mcounterstop 寄存器信息。

B2.19 mtime、mtimecmp 和 msip

为了理解这 3 个寄存器，请先参见第 13 章，系统地了解中断和异常的相关信息。

RISC-V 架构定义了一个 64 位的计时器，该计时器的值实时反映在 mtime 寄存器中，且该计时器可以通过 mtimecmp 寄存器配置其比较值，从而产生中断。注意：RISC-V 架构没有将 mtime 和 mtimecmp 寄存器定义为 CSR 寄存器，而是定义为存储器地址映射（Memory Address Mapped）的系统寄存器，具体的存储器映射地址 RISC-V 架构并没有规定，而是交由 SoC 系统集成者实现。

RISC-V 架构定义了一种软件中断，可以通过软件写 1 至 msip 寄存器来触发。有关软件中断的信息请参见第 13.3.1 节。注意：此处的 msip 寄存器和 mip 寄存器中的 MSIP 域命名不可混淆，且 RISC-V 架构并没有定义 msip 寄存器为 CSR 寄存器，而是定义为存储器地址映射的系统寄存器， RISC-V 架构并没有规定具体的存储器映射地址，而是交由 SoC 系统集成者实现。

在蜂鸟 E200 处理器的实现中，mtime/mtimecmp/msip 均由 CLINT 模块实现，有关蜂鸟 E200 的 CLINT 实现要点以及 mtime/mtimecmp/msip 分配的存储器地址区间，请参见第 13.5.5 节。

考虑到计时器计数会消耗某些动态功耗，在蜂鸟 E200 处理器的实现中，在自定义 mcounterstop 寄存器中额外增加了一位控制域。软件可以配置此控制域将 mtime 对应的计时器停止计数，从而在不需要之时停止计时器，达到省电的作用。请参见附录 B3.1 节了解更多 mcounterstop 寄存器信息。

B3 蜂鸟 E200 自定义 CSR

本节介绍蜂鸟 E200 自定义的 CSR 寄存器。

mcounterstop

考虑到 mtime、mcycle、mcycleh、minstret 和 minstreth 计数器计数会消耗某些动态功耗，因此在蜂鸟 E200 处理器的实现中，自定义此 mcounterstop 寄存器，用于控制不同计数器的运行和停止。

mcounterstop 寄存器中各控制位域如表 B-2 所示。

表 B-2　mcounterstop 寄存器各控制位

域	位	描　述
CYCLE	0	此位控制 mcycle 和 mcycleh 对应的计数器： • 如果此位为 1，则将计数器停止计数 • 如果此位为 0，则计数器正常工作 此位上电复位默认值为 0
TIMER	1	此位控制 mtime 对应的计数器： • 如果此位为 1，则将计数器停止计数 • 如果此位为 0，则计数器正常工作 此位上电复位默认值为 0
INSTRET	2	此位控制 minstret 和 minstreth 对应的计数器： • 如果此位为 1，则将计数器停止计数 • 如果此位为 0，则计数器正常工作 此位上电复位默认值为 0
Reserved	3～31	其他未使用的域为常数 0

附录C RISC-V架构的PLIC介绍

附录对于 PLIC 的介绍翻译自 RISC-V 的“特权架构文档”，本书对相关内容进行了重新组织，以求通俗易懂。

注意：附录仅介绍平台级别中断控制器（Platform Level Interrupt Controller，PLIC）部分，而 PLIC 仅是 RISC-V 整个中断机制中的一个子环节。为了更好地理解附录，请先参见第 13 章，系统地了解中断和异常的相关信息。

C1 概述

如第 13.3.1 节所述，RISC-V 架构定义了一个 PLIC 用于对多个外部中断源按优先级进行仲裁和分发。PLIC 的逻辑结构如图 C-1 所示，相关概念如下。

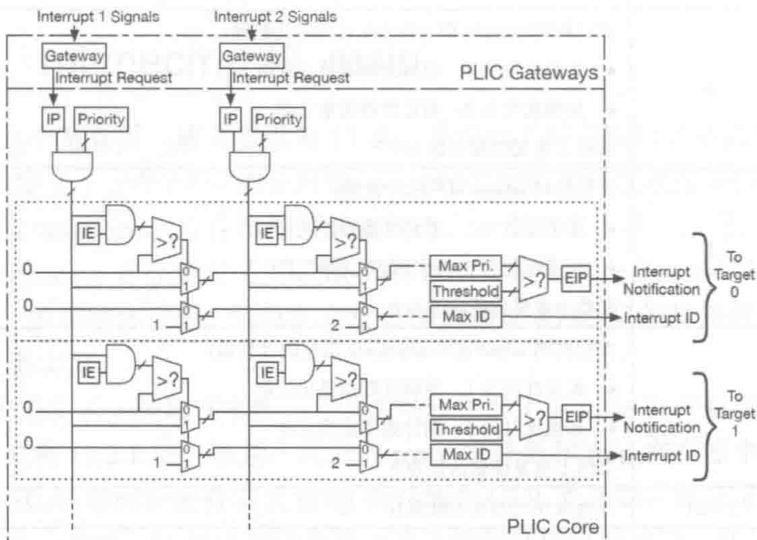


图 C-1 PLIC 逻辑结构示意图

- PLIC 中断目标
- PLIC 中断目标的阈值

- PLIC 中断源
- PLIC 中断源的闸口
- PLIC 中断源的编号
- PLIC 中断源的优先级
- PLIC 中断源的使能
- PLIC 中断通知机制
- PLIC 中断响应机制
- PLIC 中断完成机制
- PLIC 中断完整流程

下文将分别予以详述。

注意：

- 图 C-1 仅为 PLIC 的逻辑示意图，并非其真正的硬件结构图。处理器设计人员可以采取更高效的硬件设计结构予以实现。
- 图 C-1 中有两个中断目标（Target 0 和 Target 1），但 PLIC 理论上可以支持一个或者任意多个中断目标。下一节将对中断目标予以详述。
- 图 C-1 中的 IP（Interrupt Pending）表示中断源的等待标志寄存器；Priority 表示中断源的优先级寄存器；中断使能（Interrupt Enable，IE）为中断源对应于中断目标的使能寄存器；Threshold 为中断目标的优先级阈值寄存器；EIP 为发往中断目标的中断信号线。附录 C2 和 C3 将对各概念及寄存器予以详述。

C2 PLIC 中断目标

如上一节所述，PLIC 理论上可以支持一个或者任意多个中断目标（Interrupt Target），硬件设计人员可以选择具体的中断目标个数上限。

RISC-V 架构规定，PLIC 的中断目标通常是 RISC-V 架构的一个特定模式下的 Hart，有关 Hart 的概念请参见附录 A9。但是，理论上 PLIC 不仅可以用于向 RISC-V 的 Hart 发送中断，也可以向系统的其他组件发送中断（譬如 DMA、DSP 等）。

通常情况下，RISC-V 架构的 Hart 需要进入机器模式（Machine Mode）响应中断，但是 RISC-V 架构也运行低级别的工作模式（譬如用户模式）直接响应中断，此特性由 CSR 寄存器 mideleg 控制。因此对于一个 Hart 而言，其机器模式可以作为中断目标，还可以有其他模式作为中断目标。

注意：mideleg 寄存器只有在支持多种工作模式的 RISC-V 处理器中才使用。由于附录着重介绍只支持机器模式（Machine Mode Only）的架构，因此对 mideleg 寄存器不做介绍，

感兴趣的读者请参见 RISC-V“特权架构文档”原文。

如图 C-2 所示，该 PLIC 服务于 3 个 RISC-V Hart。Hart 0 有 M/U 两种模式，Hart 1 有 M/S/U 这 3 种模式，Hart 2 有 M/S/U 这 3 种模式，因此该 PLIC 总共有编号 0～7 共 8 个中断目标。

Target	Hart	Mode
0	0	M
1	0	U
2	1	M
3	1	S
4	1	U
5	2	M
6	2	S
7	2	U

图 C-2　PLIC 中断目标示例

注意：如附录 A9 所述，由于蜂鸟 E200 是单核处理器，且没有实现任何硬件超线程的技术，因此一个蜂鸟 E200 处理器核即为一个 Hart，且蜂鸟 E200 处理器核只支持机器模式。蜂鸟 E200 系统中的 PLIC 仅只有一个中断目标，Hart 0 的 M Mode。

PLIC 中断目标之阈值

如图 C-1 所示，PLIC 的每个中断目标均可以设置特定的优先级阈值（Threshold），只有中断源的优先级高于此阈值，中断才能够被发送给中断目标。有关中断源的优先级概念将在附录 C3.3 节予以详述。

中断目标的优先级阈值寄存器应该是存储器地址映射（Memory Address Mapped）的可读可写寄存器，从而软件可以通过编程配置不同的阈值来屏蔽比阈值低优先级的中断源。

C3 PLIC 中断源

如图 C-1 所示，PLIC 理论上可以支持任意多个（具体硬件实现可以选择其支持的上限）中断源（Interrupt Source）。每个中断源可以是不同的触发类型，譬如电平触发（Level-triggered）或者边沿触发（Edge-triggered）等。

PLIC 为每个中断源分配了如下功能组件和参数。

- 闸口（Gateway）和 IP
- 编号（ID）
- 优先级（Priority）
- 使能（Enable）

C3.1 PLIC 中断源之闸口（Gateway）和 IP

如图 C-1 所示，PLIC 为每个中断源分配了一个闸口（Gateway），每个闸口都有对应的中断等待寄存器，其功能如下。

- 闸口将不同触发类型的外部中断转换成统一的内部中断请求。

- 对于同一个中断源而言，闸口保证一次只发送一个中断请求（Interrupt Requst）。如图 C-1 所示，中断请求经过闸口发送后，硬件将会自动将对应的 IP 寄存器置高。
- 闸口发送一个中断请求后则启动屏蔽，如果此中断没有被处理完成，则后续的中断将会被闸口屏蔽住。有关中断完成机制，请参见附录 C4.3 节中的详细介绍。

C3.2 PLIC 中断源之编号（ID）

PLIC 为每个中断源分配了一个独一无二的编号（ID）。ID 编号 0 被预留，作为表示“不存在的中断”，因此有效的中断 ID 从 1 开始。

譬如，假设某 PLIC 的硬件实现支持 1024 个 ID，则 ID 应为 0～1023。其中，除了 0 被预留表示“不存在的中断”之外，编号 1～1023 对应的中断源接口信号线可以用于连接有效的外部中断源。

C3.3 PLIC 中断源之优先级（Priority）

如图 C-1 所示，PLIC 的每个中断源均可以设置特定的优先级，其要点如下。

- 每个中断源的优先级寄存器应该是存储器地址映射的可读可写寄存器，从而使得软件可以对其编程配置不同的优先级。
- PLIC 架构理论上可以支持任意多个优先级，硬件实现时可以选择具体的优先级个数。譬如，假设硬件实现时选择优先级寄存器的有效位为 3 位，则其可以支持的优先级个数为 0～7 这 8 个优先级。
- 优先级的数字越大，则表示优先级越高。
- 优先级 0 意味着“不可能中断”，相当于将此中断源屏蔽。

  这是因为 PLIC 的每个中断目标均可以设置特定的优先级阈值，只有中断源的优先级高于此阈值，中断才能够被发送给中断目标（见附录 C2.1 节）。由于阈值最小也为 0，因此中断源的优先级为 0 则不可能高于任何设定的阈值，即意味着“不可能中断”。

C3.4 PLIC 中断源之中断使能（Enable）

如图 C-1 所示，PLIC 为每个中断目标的每个中断源均分配了一个中断使能寄存器，其要点如下。

IE 寄存器应该是存储器地址映射的可读可写寄存器，从而使得软件可以对其编程。

- 如果 IE 寄存器被编程配置为 0，则意味着此中断源对应此中断目标被屏蔽。
- 如果 IE 寄存器被编程配置为 1，则意味着此中断源对应此中断目标被打开。

C4 PLIC 中断处理机制

C4.1 PLIC 中断通知机制（Notification）

如图 C-1 所示，对于每个中断目标而言，PLIC 对其所有中断源进行仲裁选择的原则如下。

- 对于每个中断目标来说，只有满足下列所有条件的中断源才能参与仲裁。

中断源对于该中断目标的使能（IE 寄存器）必须为 1。

中断源的优先级（优先级寄存器的值）必须大于 0。

中断源必须经过了闸口发送（IP 寄存器的值为 1）。

- 从所有参与仲裁的中断源中选择优先级最高的中断源，作为仲裁结果。如果参与仲裁的多个中断源具有相同的优先级，仲裁时则选择 ID 数目最小的中断源。
- 如果仲裁出的中断源优先级高于中断目标的优先级阈值，则产生最终的中断通知（Notification），否则不产生最终中断通知。

经过仲裁之后，如果对中断目标产生中断通知，则向该中断目标生成一根电平触发的中断线。若中断目标是一个 RISC-V Hart 的 M Mode，则该中断线的值将会反映在其 CSR 寄存器 mip 中的 MEIP 域。有关 mip 寄存器的细节，请参见附录 B2.15 节。

C4.2 PLIC 中断响应机制（Claim）

对于每个中断目标而言，如果收到了中断通知，且决定对该中断进行响应，则需要向 PLIC 发送中断响应（Interrupt Claim）消息。PLIC 定义的中断响应机制如下。

- PLIC 实现一个存储器地址映射的可读寄存器，中断目标可以通过对此寄存器进行读操作，达到中断响应的目的。作为反馈（Claim Response），此读操作将返回一个 ID，表示当前仲裁出的中断源对应的中断 ID。中断目标可以通过此 ID 得知其需要响应的具体外部中断源，如果返回的中断 ID 为 0，则表示无中断请求。
- PLIC 接收到中断响应的寄存器读操作，且返回了中断 ID 之后，硬件自动将对应中断源的 IP 寄存器清 0。

  注意：此中断源的 IP 寄存器清 0 后，其他中断源仍可以重新进行仲裁，选出下一个最高优先级的中断源，因此 PLIC 有可能会继续向该中断目标发送新的中断通知。
- 中断目标可以将该中断目标的优先级阈值设置到最大，即屏蔽掉所有的中断通知。但是该中断目标仍然可以对 PLIC 发起中断响应的寄存器读操作，PLIC 依然会返回当前仲裁出的中断源对应的中断 ID。

C4.3 PLIC 中断完成机制（Completion）

对于中断目标而言，如果彻底完成了某个中断源的中断处理操作，则需要向 PLIC 发送中断完成（Interrupt Completion）消息。PLIC 定义的中断完成机制如下。

- PLIC 实现一个存储器地址映射的可写寄存器，中断目标可以通过对此寄存器进行写操作达到中断完成的目的。此写操作需要写入一个中断 ID，以通知 PLIC 完成了此中断源的中断处理操作。
- PLIC 接收到中断完成的寄存器写操作后（写入中断 ID），硬件自动将对应中断源的闸口解除屏蔽。只有闸口解除屏蔽之后，此中断源才能经过闸口发起下一次中断请求（才能重新将 IP 寄存器置高）。

C4.4 PLIC 中断完整流程

综上所述，对于每个中断源的中断而言，如图 C-3 所示，其完整流程总结如下。

- 如果闸口没有被屏蔽，则中断源经过闸口发起中断请求（Interrupt Request）。闸口发送一个中断请求后：硬件自动将其对应的 IP 寄存器置高；PLIC 硬件将对应中断源的闸口启动屏蔽，后续的中断将会被闸口屏蔽住。
- 按照附录 C4.1 节所述的中断仲裁机制，如果经过 PLIC 硬件仲裁后选中了该中断源，且其优先级高于中断目标的阈值，PLIC 则向中断目标发起中断通知（Interrupt Notification）。
- 中断目标收到中断通知后，如果决定响应此中断，则使用软件向 PLIC 发起中断响应的读操作。作为响应反馈，PLIC 返回该中断源的中断 ID。同时，硬件自动将其对应的 IP 寄存器清零。
- 中断目标收到中断 ID 之后，可以通过此 ID 得知其需要响应的具体外部中断源。然后进入该外部中断源对应的具体中断服务程序（Interrupt Service Routine）中进行处理。
- 待彻底完成了中断处理之后，中断目标使用软件向 PLIC 发起“中断完成”的写操作，写入要完成的中断 ID。同时，PLIC 硬件将对应中断源的闸口解除屏蔽，允许其能够发起下一次新的中断请求。

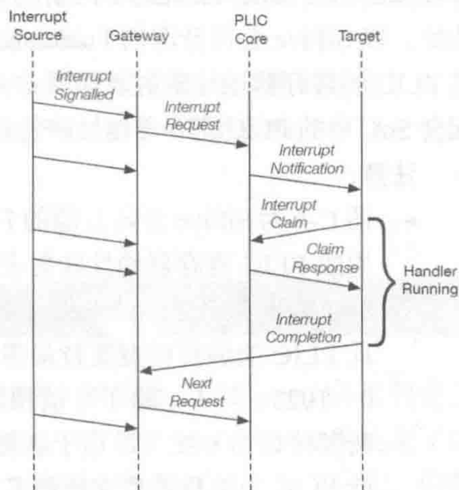


图 C-3 PLIC 中断完整流程

C5 PLIC 寄存器总结

综上所述，PLIC 需要支持的若干种存储器地址映射的寄存器如下。

- 每个中断源的中断等待（Interrupt Pending，IP）寄存器（只读）。
- 每个中断源的优先级寄存器（可读可写）。
- 每个中断目标对应每个中断源的中断使能寄存器（可读可写）。
- 每个中断目标的阈值寄存器（可读可写）。
- 每个中断目标的中断响应寄存器（可读）。
- 每个中断目标的中断完成寄存器（可写）。

RISC-V 架构文档中并没有对上述寄存器定义明确的存储器地址，而是交给硬件实现者自定义。因此硬件设计人员可以按照所处 SoC 系统的不同情况分配具体的存储器映射地址。以 SiFive 公司开源的 Freedom E310 SoC 平台为例，其 PLIC 的寄存器地址映射表如图 C-4 所示。有关蜂鸟 E200 配套 SoC 中的 PLIC 寄存器地址映射表，请参见第 13.5.6 节。

Address	Description
0x0C00_0000	*Reserved*
0x0C00_0004	source 1 priority
0x0C00_0008	source 2 priority
...	
0x0C00_0FFC	source 1023 priority
0x0C00_1000	Start of pending array
...	(read-only)
0x0C00_107C	End of pending array
0x0C00_1800	
...	*Reserved*
0x0C00_1FFF	
0x0C00_2000	target 0 enables
0x0C00_2080	target 1 enables
...	
0x0C1E_FF80	target 15871 enables
0x0C1F_0000	
...	*Reserved*
0x0C1F_FFFC	
0x0C20_0000	target 0 priority threshold
0x0C20_0004	target 0 claim/complete
0x0C20_1000	target 1 priority threshold
0x0C20_1004	target 1 claim/complete
...	
0x0FFF_F000	target 15871 priority threshold
0x0FFF_F004	target 15871 claim/complete

图 C-4　Freedom E310 SoC 平台的 PLIC 寄存器地址映射表

注意：

- 图 C-4 为 SiFive 公司开源的 Freedom E310 SoC 中使用的 PLIC 寄存器地址映射表。有关 Freedom E310 SoC 的更多信息，请参见第 18.1 节。

  此 PLIC 的编程模型支持最多 1024 个 ID，则 ID 为 0～1023。其中，除了 0 被预留表示“不存在的中断”之外，编号 1～1023 对应的中断源接口信号线可以用于连接有效的外部中断源。

  此 PLIC 的编程模型支持最多 15872 个中断目标（Target 0～Target 15871）。
- 图 C-4 中的“source 1 priority”～“source 1023 priority”对应每个中断源的优先级寄存器（可读可写）。虽然每个优先级寄存器对应于一个 32 位的地址区间（4 个字节），但是优先级寄存器的有效位可以只有几位（其他位固定为 0 值）。假设硬件实现优先级寄存器的有效位为 3 位，则其可以支持的优先级个数为 0～7 这 8 个优先级。
- 图 C-4 中的“Start of pending array”～“End of pending array”对应每个中断源的 IP 中断等待寄存器（只读）。由于每个中断源的 IP 仅有一位宽，而每个寄存器对应于一个 32 位的地址区间（4 个字节），因此每个寄存器可以包含 32 个中断源的 IP。按照此规则，譬如“Start of pending array”寄存器包含中断源 0～31 的 IP 寄存器值，其他依次类推。每 32 个中断源的 IP 被组织在一个寄存器中，总共 1024 个中断源则

需要 32 个寄存器，其地址为 0x0C00_1000～0x0C00_107C 的 32 个地址。

- 图 C-4 中的“target 0 enables”对应每个中断源的中断使能寄存器（可读可写）。与 IP 寄存器同理，由于每个中断源的 IE 仅有一位宽，而每个寄存器对应于一个 32 位的地址区间（4 个字节），因此每个寄存器可以包含 32 个中断源的 IE。

  按照此规则，对于“target 0”而言，每 32 个中断源的 IE 被组织在一个寄存器中，总共 1024 个中断源，则需要 32 个寄存器，其地址为 0x0C00_2000～0x0C00_207C 的 32 个地址区间。
- 图 C-4 中的“target 1 enables”～“target 15871 enables”与上述“target 0 enables”同理，每一个“target”占据 32 个地址区间。
- 图 C-4 中的“target 0 priority threshold”“target 15871 priority threshold”对应每个中断目标的阈值寄存器（可读可写）。

  虽然每个阈值寄存器对应于一个 32 位的地址区间（4 个字节），但是阈值寄存器的有效位个数应该与每个中断源的优先级寄存器有效位个数相同。
- 图 C-4 中的“target 0 claim/complete”～“target 15871 claim/complete”对应每个中断目标的“中断响应”寄存器和“中断完成”寄存器。

  如附录 C4.2 节和 C4.3 节中所述，对于每个中断目标而言，由于“中断响应”寄存器为可读，“中断完成”寄存器为可写，因此将其合并作为一个寄存器共享同一个地址，成为一个可读可写的寄存器。

C6 总结与比较

对 ARM 的 Cortex-M 或 Cortex-A 系列比较熟悉的读者，想必会了解 Cortex-M 系列定义的嵌套向量中断控制器（Nested Vector Interrupt Controller，NVIC）和 Cortex-A 系列定义的通用中断控制器（General Interrupt Controller，GIC）。这两种中断控制器都非常强大，但是功能也相对非常复杂。

相比而言，RISC-V 架构定义的 PLIC 则非常简单，这反映了 RISC-V 架构力图简化硬件的设计哲学。此外，RISC-V 架构也允许处理器设计者定义其自有的中断控制器，因此可以从很多开源或商用的 RISC-V 处理器 IP 中看到其他非标准的中断控制器身影，本书在此不一一列举。

附录D　存储器模型背景介绍

附录将对存储器模型（Memory Model）的相关背景知识进行简介。请注意，由于存储器模型是计算机体系结构中非常晦涩的一个概念，本书作为一本通俗读本，重在力图做到通俗易懂。因此，对于存储器模型的介绍难免有失之学术精准之处，关于其更为严谨的学术定义读者可以自行查阅其他资料进行了解。

D1　为何要有存储器模型的概念

本节先介绍为何要有存储器模型（Memory Model）这个概念？即，存储器模型要解决什么问题？

在最早期的处理器设计时代，处理器都是单核。只有一个处理器单核执行软件程序时，在单核中处理对于存储器读写指令的执行很好理解，也就是说，处理器对于存储器读写操作的结果严格和程序顺序（Program-Order）定义的结果一致。程序顺序定义的结果，就是指处理器严格按照顺序逐条地执行其汇编指令的结果。

理论上来讲，对于存储器访问地址有相关性的指令（譬如前一条指令写某个存储器地址，之后另一条指令读该存储器地址），那么它们的执行顺序一定不能被颠倒，否则会造成结果错误。而对于存储器访问地址没有相关性的指令（譬如前一条指令写某个存储器地址，之后另一条指令读另外一个不同的存储器地址），那么它们的执行顺序可以被颠倒，不会影响最终的执行结果，不会造成结果错误。

基于上述的原理，一方面编译器可以对程序生成的汇编指令流中的指令顺序进行适当改变，从而在某些情况下优化性能（譬如将某些有数据相关性的指令中间插入后序没有数据相关性的指令）；另一方面，处理器核的硬件在执行程序时也可以动态地调整指令的执行顺序，从而提高处理器的执行性能。

但是，随着技术的进步和发展，处理器设计进入多核时代，情况变得微妙起来。假设不同的处理器核需要同时访问共享的存储器区间，对共享的数据区间进行读写。由于不同的处理器核在执行程序时存在着很多种随机性和不确定性，因此它们访问到共享存储器区间的先后顺序也存在着随机性和不确定性，从而造成多核程序的执行结果不可确知。这种不可确知

性就会给软件开发造成困扰，对运行多核程序的系统造成不稳定性。

在第 1.1.1 节中介绍过，指令集架构（ISA）是衔接底层硬件和高层软件之间的一个抽象层，该抽象层定义了任何软件程序员需要了解的硬件信息。为了能够给上层软件明确地规定清楚多核程序访问共享数据的结果，在指令集架构中便引入了存储器模型的概念。

D2 存储器模型定义了什么

存储器模型（Memory Model），又称存储器一致性模型（Memory Consistency Model），用于定义系统中对存储器访问需要遵守的规则。只要软件和硬件都明确遵循存储器模型定义的规则，就可以保证多核程序也能够运行得到确切的结果。

存储器模型往往是现代 ISA 很重要的一部分，因此使用高级语言的程序员、设计编译器的软件工程师、处理器硬件设计人员都需要了解其所使用 ISA 的存储器模型。

下面以最有代表性的 3 种存储器模型——按序一致性模型（Sequential Consistency Model）、松散一致性模型（Relaxed Consistency Model）和释放一致性模型（Release Consistency Model）为例加以介绍以利于读者理解。

D2.1 按序一致性模型

按序一致性模型（Sequential Consistency Model），顾名思义就是“严格按序”模型。如果处理器的指令集架构符合按序一致性模型，那么在多个处理器核上执行的程序就好像在一个单核处理器上顺序执行一样。例如系统有两个处理器核，分别是 Core 0 和 Core 1。Core 0 执行了 A、B、C、D 共 4 条存储器访问指令，Core 1 执行了 a、b、c、d 共 4 条存储器访问指令。对于程序员而言，按序一致性模型的系统上执行这 8 条指令的效果就好像在一个 Core 上顺序执行了 A、a、B、b、C、c、D、d 的指令流，或者是 A、B、a、b、C、c、D、d，还可以是 A、B、C、D、a、b、c、d。总之，只要同时符合 Core 0 和 Core 1 的程序顺序（即单独从 Core 0 的角度看，其程序顺序必须是 A→B→C→D，单独从 Core 1 的角度看，其程序顺序必须是 a→b→c→d）的任意组合，都是合法的组合。

综上，我们可以总结出按序一致性模型的两条规则。

- 各个处理器核按照其程序顺序来执行程序，执行完一条后，启动执行下一条指令，不能够改变存储器访问指令的顺序（即便访问的是不同的存储器地址）。
- 从全局来看，每一个存储器写指令的操作都需要能够被系统中的所有处理器核同时观测到。就好像处理器系统（包括所有的处理器核）和存储系统之间有一个开关，一次只会连接一个处理器核和存储系统，因此对存储器的访问都是原子的、串行化的。

按序一致性模型是最简单和直观的存储器模型，但这也限制了 CPU 硬件和编译器的优化，从而影响了整个系统的性能，于是便有了松散一致性模型。

D2.2 松散一致性模型

松散一致性模型（Relaxed Consistency Model），顾名思义就是“松散”模型。如附录 D1 中所述，对于不同存储器地址的访问指令，对单核而言，理论上是可以改变其执行顺序的。松散一致性模型允许在多核系统中的每个单核改变其存储器访问指令（必须访问的是不同的地址）的执行顺序。

松散一致性模型由于解除了束缚，系统的运行性能更加好。但多核程序这样无所束缚地执行的结果会变得完全不可确知，为了能够限定处理器的执行顺序，便引入了特殊的存储器屏障（Memory FENCE）指令。FENCE 指令用于屏障“数据”存储器访问的执行顺序，如果在程序中添加了一条 FENCE 指令，则该 FENCE 指令能够保证“在 FENCE 之前所有指令进行的数据访存结果”必须比“在 FENCE 之后所有指令进行的数据访存结果”先被观测到。通俗地讲，FENCE 指令就像一堵屏障一样，在 FENCE 指令之前的所有数据存储器访问指令，必须比该 FENCE 指令之后的所有数据存储器访问指令先执行。

通过将松散一致性模型和存储器屏障指令相结合，便可以达到性能和功能的平衡。譬如，在不关心存储器访问顺序的场景下可以达到高的运行性能，而在某些关心存储器访问顺序的场景下，软件程序员可以明确使用存储器屏障指令来约束指令的执行顺序。

D2.3 释放一致性模型

“释放一致性模型（Release Consistency Model）”进一步支持获取-释放（Acquire-Release）机制，其核心要点如下。

- 定义一种获取（Release）指令，它仅屏障其之前的所有存储器访问操作。
- 定义一种释放（Acquire）指令，它仅屏障其之后的所有存储器访问操作。

由于获取和释放指令仅屏障一个方向，因此相比 FENCE 指令更加松散。

附录 D3 将结合一个具体的应用实例帮助读者进一步理解获取-释放机制和上述不同模型的差异。

D2.4 存储器模型总结

在上述的介绍中，为了通俗易懂，我们以处理器核为单位介绍了存储器模型的概念，强调了存储器模型在多核系统中的重要性。

但是，现今的处理器设计技术突飞猛进，早已经突破了多核的概念，在一个处理器核中

设计多个硬件线程的技术也早已成熟。譬如硬件超线程（Hyper-threading）技术，便是在一个处理器核中实现多份硬件线程，每套线程有自己独立的寄存器组等上下文相关的资源，但是大多数的运算资源被所有硬件线程复用，因此面积效率很高。在这样的硬件超线程处理器中，一个核内的多个线程同样存在着与多核系统类似的存储器模型问题。

并且经过多年的发展，除了本附录介绍的 3 种模型之外，存储器模型的种类已经发展出众多不同的模型，本书限于篇幅，在此不一一列举，感兴趣的读者请自行查阅。

D3 存储器模型应用实例

在多核软件开发中经常有需要进行同步（Synchronization）的场景，一个需要进行同步的典型双核场景如下为：Core 0 要写入一片数据到某一段地址区间中，然后通知 Core 1 将此片数据读走。

为了完成上述功能，程序员开发了一个多核应用程序，预期如下。

- Core 0 和 Core 1 二者约定了一个共享的全局变量作为旗语。程序的全局变量在硬件上的本质是在存储器中分配一个地址保存该变量的值，Core 0 和 Core 1 都能够访问到该地址。
- Core 0 完成了写数据操作之后，便将此共享变量写为一个“特殊的数值”。
- Core 1 则不断地在监测此共享变量的值，一旦其监测到了“特殊的数值”，便认为可以安全地将数据从地址区间中读走。

Core 0 和 Core 1 的程序可以被抽象如下。

- Core 0：写入数据→设置旗语。
- Core 1：监测旗语→监测到旗语的“特殊的数值”→读取数据。

从上述描述可以看出，为了能够准确地实现交互数据的功能，Core 0 的“写入数据”和“设置旗语”指令的执行顺序一定不能发生改变；同样，Core 1 的“监测旗语”和“读取数据”指令的执行顺序也一定不能发生改变。

在使用按序一致性模型的多核系统中，执行顺序一定能够得到保证，因此程序执行的结果能够满足程序员的预期。

但是在使用松散一致性模型的系统中，由于“数据”和“旗语”所处的存储器地址不一样，理论上是可以改变其执行顺序的。因此编译器或者处理器硬件本身可能会进行优化，使得程序最终的执行结果可能并不是程序员期望的那样。在松散一致性模型的系统中，必须要在程序中插入存储器屏障（Memory FENCE）指令，抽象如下。

- Core 0：写入数据→插入 FENCE 指令→设置旗语。
- Core 1：监测旗语→监测到旗语的“特殊的数值”→插入 FENCE 指令→读取数据。

由于 FENCE 指令能够将其前后的存储器访问指令屏障开来而不会发生执行顺序的改变，因此能够保证程序执行的结果满足程序员的预期。

但是经过进一步深入观察可以发现如下规律。

- 如果有一条指令能够将“插入FENCE指令”和“设置旗语”合二为一，那么理论上只需要屏障其之前的存储器访问操作即可（无须屏障其之后的操作）。
- 同理，如果有一条指令能够将“监测旗语”和“插入FENCE指令”合二为一，那么理论上只需要屏障其之后的存储器访问操作即可（而无须屏障其之前的操作）。
- 假设能够做到上述两点，由于只需要屏蔽一个方向，可以进一步提高性能。

因此为了能够再进一步地提高性能，可以使用释放一致性模型中的获取-释放的机制，软件便可以进一步改写如下。

- Core 0：写入数据→释放旗语。
- Core 1：获取旗语→获取旗语发现“特殊的数字”→读取数据。

由于释放操作屏障了其之前的存储器访问指令，获取操作屏障了其之后的存储器访问指令，因此同样可以保证程序执行的结果满足程序员的预期。

至此，上述问题终于得到了完美的解决！

D4 RISC-V架构的存储器模型

如附录 D2.4 节中所述，存储器模型不仅适用于多核场景，也适用于多线程场景。在描述存储器模型时，如果笼统地使用“处理器核”的概念进行描述会有失精确。因此如附录 A9 中所述，在 RISC-V 的架构文档中严谨地定义了一个 Hart 的概念，表示一个硬件线程。

RISC-V 架构明确规定在不同 Hart 之间使用松散一致性模型，并相应地定义了存储器屏障指令（FENCE 和 FENCE.I），用于屏障存储器访问的顺序。另外，RISC-V 架构定义了可选的（非必需的）存储器原子操作指令（A 扩展指令子集），可进一步支持释放一致性模型，请参见附录 A14.5 节了解更多相关信息。

附录 E 存储器原子操作指令背景介绍

附录将结合多线程“锁”的示例对存储器原子操作指令的应用背景进行简介。请注意，由于“锁”是多线程编程中比较晦涩的一个概念，本书作为一本通俗读本，重在力图做到通俗易懂，因此，对于“锁”的介绍难免有失之学术精准之处，关于其更为严谨的学术定义读者可以自行查阅其他资料进行了解。

E1 什么是“上锁”问题

在多核软件开发中经常有需要进行“上锁”的场景，此处的“锁”是指软件中定义的功能命名，多核软件中存在着多种不同的锁（譬如 spin_lock 和 mutex_lock 等）。一个需要进行“上锁”的典型三核场景如 Core 0、Core 1 和 Core 2 共享一片数据区间，但是一个时间只有一个核（Core）能够独占此数据区间，因此 Core 0、Core 1 和 Core 2 需要竞争，竞争的策略如下。

- Core 0、Core 1 和 Core 2 三者约定了一个共享的全局变量作为“锁”。

程序的全局变量在硬件上的本质是在存储器中分配一个地址保存该变量的值，Core 0、Core 1 和 Core 2 都能够访问到该地址。

锁中的值为 0 表示当前共享数据区空闲，没有被任何一个核独占。

锁中的值为 1 表示当前共享数据区被某个核独占。

- 当某个核每次独占共享数据区完成了相关的操作后，便会释放数据区，通过向锁中写入数值 0 将其释放。
- 没有独占数据区的核（譬如 Core 0 独占时，Core 1 和 Core 2）都会不断地读锁中的值，判别其是否空闲。一旦发现锁空闲，便会向锁中写入数值 1 进行“上锁”，试图将共享数据区进行独占。

如果使用普通的读（Load）和写（Store）指令分别对存储器进行读和写操作，那么第一次读（发现锁空闲）和下一次写（写入数值 1 上锁）之间存在着时间差，并且是两次分立的操作，不同的核发出的读写操作可能彼此交织在一起，那么可能出现下述这种情况。

- 当数据区空闲之后，两个核（Core 1 和 Core 2）均读到了锁的值为 0，于是认为自己可以独占数据区，并向锁中写入数值 1。
- 按照规则，只能有一个核能够独占此共享区，但是此时两个核却都以为自己取得了

共享区的独占权，从而造成程序的运行结果变得不正确。

E2 通过原子操作解决“上锁”问题

上一节介绍了多核“上锁”时面临的竞争问题。为了解决该问题，如果能够引入一种“原子”操作，让第一次读（发现锁空闲）和下一次写（写入数值 1）操作成为一个完整的整体，期间不被其他核的访问所打断，那么便可以保证一次只能有一个核上锁成功。

为了支持“原子”操作，以 ARM 指令集架构为例，ARM 架构早期引入了原子交换（SWP）指令。该指令同时将存储器中的值读出至结果寄存器，并将另一个源操作数的值写入存储器中相同的地址，实现通用寄存器中的值和存储器中的值的交换。并且，在第一次读操作之后，硬件便将总线或者目标存储器锁定，直到第二次写操作完成之后才解锁，期间不允许其他的核访问，这便是在 AHB 总线中开始引入“Lock”信号支持总线锁定功能的由来。

有了 SWP 指令和硬件锁定总线功能的支持，每个核便可以使用 SWP 指令进行上锁，步骤如下。

- 步骤 1：使用 SWP 指令将锁中的值读出，并向锁中写入数值 1。该过程为一个整体性的原子操作，读和写操作之间其他核不会访问到锁。
- 步骤 2：对读取的值进行判断，如果发现锁中的值为 1，则意味着当前锁正在被其他的核占用，上锁失败，因此继续回到步骤 1 重复再读；如果发现锁中的值为 0，则意味着当前锁已经空闲，同时由于 SWP 指令也以原子操作的方式向其写入了数值 1，则上锁成功，可以进行独占。

原子指令操作除了解决上锁问题之外，还可以解决很多其他的问题，本书在此不做一一赘述。

E3 通过互斥操作解决“上锁”问题

上一节介绍了使用原子操作指令解决多核“上锁”时面临的竞争问题，但是“原子操作”指令也存在着弊端。它会将总线锁定住，导致其他的核无法访问总线，在核数众多且频繁抢锁的场景下，会造成总线长期被锁的情况，严重影响系统的运行性能。

因此 ARM 架构之后又引入了一种新的互斥（Exclusive）类型的存储器访问指令来替代 SWP 指令，其核心要义可以简述如下。

- 定义一种互斥读（Load-Exclusive）指令。该指令与普通的读指令类似，向存储器进行一次次读操作。
- 定义一种互斥写（Store-Exclusive）指令。该指令与普通的写指令类似，但是它不一

定能够执行成功。该指令会向其结果寄存器写回操作成功或是失败的标志信息，如果执行失败，意味着没有真正写入存储器。

- 在系统中实现一个监测器（Monitor）。该监测器能够保证只有当互斥读（Load-Exclusive）和互斥写（Store-Exclusive）指令成对地访问相同的地址，且其间隙中没有任何其他的写操作（来自于任何一个线程）访问过同样的地址，互斥写（Store-Exclusive）指令才会执行成功。

为了实现上述功能，系统中的监测器的硬件实现机理略显复杂。为了不使读者陷入理解复杂问题的泥潭，本书在此将其略过，不加详述，感兴趣的读者可以自行查阅其他资料。

- 互斥读（Load-Exclusive）指令执行的存储器读操作和互斥写（Store-Exclusive）指令执行的存储器写操作之间并不会将总线锁定，因此并不会造成系统性能的下降。这是与原子操作指令的最大不同。

为了区别出普通的读（Load）/写（Store）和互斥读（Load-Exclusive）/互斥写（Store-Exclusive）指令发起的存储器访问操作，需要特殊的信号加以指示。这也是AXI总线中引入了互斥属性信号的缘由。

有了互斥读（Load-Exclusive）指令和互斥写（Store-Exclusive）指令和系统监测器的支持，每个核便可以使用互斥读（Load-Exclusive）指令和互斥写（Store-Exclusive）指令进行上锁，其步骤如下。

- 步骤1：使用互斥读（Load-Exclusive）指令将锁中的值读出。
- 步骤2：对读取的值进行判断，如果发现锁中的值为1，意味着当前锁正在被其他的核占用，继续回到步骤1重复再读；如果发现锁中的值为0，意味着当前锁已经空闲，进入步骤3。
- 步骤3：使用互斥写（Store-Exclusive）指令向锁中写入数值1，试图对其进行上锁，然后对该指令的返回结果（成功还是失败的标志信息）进行判断。如果返回结果表示该互斥写（Store-Exclusive）指令执行成功，意味着上锁成功，否则意味着上锁失败。

由于第一次读和第二次写之间并没有将总线锁定，因此其他的核也可能访问锁。并且其他核也可能发现锁中的值为0，并继而向锁中写入数值1试图上锁，但系统中的监测器会保证只有先进行互斥写（Store-Exclusive）的核才能成功，后进行互斥写（Store-Exclusive）的核会失败，从而保证每一次只能有一个核成功上锁。

E4 RISC-V架构的相关指令

在RISC-V架构的基本指令集（必选的）中，并没有定义原子操作指令和互斥指令，但是在可选的“A”扩展指令子集中支持了此两种指令，请参见附录A14.5节了解其具体的指令信息。

附录 F RISC-V 指令编码列表

附录截取自 RISC-V“指令集文档”（riscv-spec-v2.2.pdf），以便于读者快速查阅。

F1 RV32I 指令编码

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 pred succ		00000	000	00000	0001111	FENCE
0000 0000 0000		00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

Environment Call and Breakpoint

000000000000	00000	000	00000	1110011	ECALL
000000000001	00000	000	00000	1110011	EBREAK

Trap-Return Instructions

0000000	00010	00000	000	00000	1110011	URET
0001000	00010	00000	000	00000	1110011	SRET
0011000	00010	00000	000	00000	1110011	MRET

Interrupt-Management Instructions

0001000	00101	00000	000	00000	1110011	WFI

F2 RV32M 指令编码

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

F3 RV32A 指令编码

RV32A Standard Extension

00010	aq	rl	00000	rs1	010	rd	0101111	LR.W
00011	aq	rl	rs2	rs1	010	rd	0101111	SC.W
00001	aq	rl	rs2	rs1	010	rd	0101111	AMOSWAP.W
00000	aq	rl	rs2	rs1	010	rd	0101111	AMOADD.W
00100	aq	rl	rs2	rs1	010	rd	0101111	AMOXOR.W
01100	aq	rl	rs2	rs1	010	rd	0101111	AMOAND.W
01000	aq	rl	rs2	rs1	010	rd	0101111	AMOOR.W
10000	aq	rl	rs2	rs1	010	rd	0101111	AMOMIN.W
10100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAX.W
11000	aq	rl	rs2	rs1	010	rd	0101111	AMOMINU.W
11100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAXU.W

F4 RV32F 指令编码

RV32F Standard Extension

imm[11:0]			rs1	010	rd	0000111	FLW
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	FSW
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.S
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.S
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.S
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.S
0000000		rs2	rs1	rm	rd	1010011	FADD.S
0000100		rs2	rs1	rm	rd	1010011	FSUB.S
0001000		rs2	rs1	rm	rd	1010011	FMUL.S
0001100		rs2	rs1	rm	rd	1010011	FDIV.S
0101100		00000	rs1	rm	rd	1010011	FSQRT.S
0010000		rs2	rs1	000	rd	1010011	FSGNJ.S
0010000		rs2	rs1	001	rd	1010011	FSGNJN.S
0010000		rs2	rs1	010	rd	1010011	FSGNJX.S
0010100		rs2	rs1	000	rd	1010011	FMIN.S
0010100		rs2	rs1	001	rd	1010011	FMAX.S
1100000		00000	rs1	rm	rd	1010011	FCVT.W.S
1100000		00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000		00000	rs1	000	rd	1010011	FMV.X.W
1010000		rs2	rs1	010	rd	1010011	FEQ.S
1010000		rs2	rs1	001	rd	1010011	FLT.S
1010000		rs2	rs1	000	rd	1010011	FLE.S
1110000		00000	rs1	001	rd	1010011	FCLASS.S
1101000		00000	rs1	rm	rd	1010011	FCVT.S.W
1101000		00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000		00000	rs1	000	rd	1010011	FMV.W.X

F5 RV32D 指令编码

RV32D Standard Extension

31–27	26–25	24–20	19–15	14–12	11–7	6–0	
imm[11:0]			rs1	011	rd	0000111	FLD
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	FSD
rs3	01	rs2	rs1	rm	rd	1000011	FMADD.D
rs3	01	rs2	rs1	rm	rd	1000111	FMSUB.D
rs3	01	rs2	rs1	rm	rd	1001011	FNMSUB.D
rs3	01	rs2	rs1	rm	rd	1001111	FNMADD.D
0000001		rs2	rs1	rm	rd	1010011	FADD.D
0000101		rs2	rs1	rm	rd	1010011	FSUB.D
0001001		rs2	rs1	rm	rd	1010011	FMUL.D
0001101		rs2	rs1	rm	rd	1010011	FDIV.D
0101101		00000	rs1	rm	rd	1010011	FSQRT.D
0010001		rs2	rs1	000	rd	1010011	FSGNJ.D
0010001		rs2	rs1	001	rd	1010011	FSGNJN.D
0010001		rs2	rs1	010	rd	1010011	FSGNJX.D
0010101		rs2	rs1	000	rd	1010011	FMIN.D
0010101		rs2	rs1	001	rd	1010011	FMAX.D
0100000		00001	rs1	rm	rd	1010011	FCVT.S.D
0100001		00000	rs1	rm	rd	1010011	FCVT.D.S
1010001		rs2	rs1	010	rd	1010011	FEQ.D
1010001		rs2	rs1	001	rd	1010011	FLT.D
1010001		rs2	rs1	000	rd	1010011	FLE.D
1110001		00000	rs1	001	rd	1010011	FCLASS.D
1100001		00000	rs1	rm	rd	1010011	FCVT.W.D
1100001		00001	rs1	rm	rd	1010011	FCVT.WU.D
1101001		00000	rs1	rm	rd	1010011	FCVT.D.W
1101001		00001	rs1	rm	rd	1010011	FCVT.D.WU

F6 RVC 指令编码

注意：在 RVC 指令中，有着不同类型的非法编码，如下表中最右侧一栏标注的。

- RES：表示这种编码被预留作为未来扩展使用。
- NES：表示这种编码被预留作为非标准扩展。
- HINT:表示这种编码被预留作为微架构的指示,硬件实现可以选择将其实现为 NOP。

15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0	
000	0			0	00	*Illegal instruction*
000	nzuimm[5:4|9:6|2|3]			rd′	00	C.ADDI4SPN *(RES, nzuimm=0)*
001	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.FLD *(RV32/64)*
001	uimm[5:4|8]	rs1′	uimm[7:6]	rd′	00	C.LQ *(RV128)*
010	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.LW
011	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.FLW *(RV32)*
011	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.LD *(RV64/128)*
100	—				00	*Reserved*
101	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.FSD *(RV32/64)*
101	uimm[5:4|8]	rs1′	uimm[7:6]	rs2′	00	C.SQ *(RV128)*
110	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.SW
111	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.FSW *(RV32)*
111	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.SD *(RV64/128)*

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzuimm[5]	rs1/rd≠0	nzuimm[4:0]	10	C.SLLI *(HINT, rd=0; RV32 NSE, nzuimm[5]=1)*
000	0	rs1/rd≠0	0	10	C.SLLI64 *(RV128; RV32/64 HINT; HINT, rd=0)*
001	uimm[5]	rd	uimm[4:3|8:6]	10	C.FLDSP *(RV32/64)*
001	uimm[5]	rd≠0	uimm[4|9:6]	10	C.LQSP *(RV128; RES, rd=0)*
010	uimm[5]	rd≠0	uimm[4:2|7:6]	10	C.LWSP *(RES, rd=0)*
011	uimm[5]	rd	uimm[4:2|7:6]	10	C.FLWSP *(RV32)*
011	uimm[5]	rd≠0	uimm[4:3|8:6]	10	C.LDSP *(RV64/128; RES, rd=0)*
100	0	rs1≠0	0	10	C.JR *(RES, rs1=0)*
100	0	rd≠0	rs2≠0	10	C.MV *(HINT, rd=0)*
100	1	0	0	10	C.EBREAK
100	1	rs1≠0	0	10	C.JALR
100	1	rs1/rd≠0	rs2≠0	10	C.ADD *(HINT, rd=0)*
101	uimm[5:3|8:6]		rs2	10	C.FSDSP *(RV32/64)*
101	uimm[5:4|9:6]		rs2	10	C.SQSP *(RV128)*
110	uimm[5:2|7:6]		rs2	10	C.SWSP
111	uimm[5:2|7:6]		rs2	10	C.FSWSP *(RV32)*
111	uimm[5:3|8:6]		rs2	10	C.SDSP *(RV64/128)*

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
000	0	0		0		01	C.NOP
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]		01	C.ADDI *(HINT, nzimm=0)*
001	imm[11|4|9:8|10|6|7|3:1|5]					01	C.JAL *(RV32)*
001	imm[5]	rs1/rd≠0		imm[4:0]		01	C.ADDIW *(RV64/128; RES, rd=0)*
010	imm[5]	rd≠0		imm[4:0]		01	C.LI *(HINT, rd=0)*
011	nzimm[9]	2		nzimm[4|6|8:7|5]		01	C.ADDI16SP *(RES, nzimm=0)*
011	nzimm[17]	rd≠{0, 2}		nzimm[16:12]		01	C.LUI *(RES, nzimm=0; HINT, rd=0)*
100	nzuimm[5]	00	rs1′/rd′	nzuimm[4:0]		01	C.SRLI *(RV32 NSE, nzuimm[5]=1)*
100	0	00	rs1′/rd′	0		01	C.SRLI64 *(RV128; RV32/64 HINT)*
100	nzuimm[5]	01	rs1′/rd′	nzuimm[4:0]		01	C.SRAI *(RV32 NSE, nzuimm[5]=1)*
100	0	01	rs1′/rd′	0		01	C.SRAI64 *(RV128; RV32/64 HINT)*
100	imm[5]	10	rs1′/rd′	imm[4:0]		01	C.ANDI
100	0	11	rs1′/rd′	00	rs2′	01	C.SUB
100	0	11	rs1′/rd′	01	rs2′	01	C.XOR
100	0	11	rs1′/rd′	10	rs2′	01	C.OR
100	0	11	rs1′/rd′	11	rs2′	01	C.AND
100	1	11	rs1′/rd′	00	rs2′	01	C.SUBW *(RV64/128; RV32 RES)*
100	1	11	rs1′/rd′	01	rs2′	01	C.ADDW *(RV64/128; RV32 RES)*
100	1	11	—	10	—	01	*Reserved*
100	1	11	—	11	—	01	*Reserved*
101	imm[11|4|9:8|10|6|7|3:1|5]					01	C.J
110	imm[8|4:3]		rs1′	imm[7:6|2:1|5]		01	C.BEQZ
111	imm[8|4:3]		rs1′	imm[7:6|2:1|5]		01	C.BNEZ

附录G RISC-V伪指令列表

附录截取自RISC-V“指令集文档”(riscv-spec-v2.2.pdf)，以便读者快速查阅。

Pseudoinstruction	Base Instruction	Meaning
rdinstret[h] rd	csrrs rd, instret[h], x0	Read instructions-retired counter
rdcycle[h] rd	csrrs rd, cycle[h], x0	Read cycle counter
rdtime[h] rd	csrrs rd, time[h], x0	Read real-time clock
csrr rd, csr	csrrs rd, csr, x0	Read CSR
csrw csr, rs	csrrw x0, csr, rs	Write CSR
csrs csr, rs	csrrs x0, csr, rs	Set bits in CSR
csrc csr, rs	csrrc x0, csr, rs	Clear bits in CSR
csrwi csr, imm	csrrwi x0, csr, imm	Write CSR, immediate
csrsi csr, imm	csrrsi x0, csr, imm	Set bits in CSR, immediate
csrci csr, imm	csrrci x0, csr, imm	Clear bits in CSR, immediate
frcsr rd	csrrs rd, fcsr, x0	Read FP control/status register
fscsr rd, rs	csrrw rd, fcsr, rs	Swap FP control/status register
fscsr rs	csrrw x0, fcsr, rs	Write FP control/status register
frrm rd	csrrs rd, frm, x0	Read FP rounding mode
fsrm rd, rs	csrrw rd, frm, rs	Swap FP rounding mode
fsrm rs	csrrw x0, frm, rs	Write FP rounding mode
fsrmi rd, imm	csrrwi rd, frm, imm	Swap FP rounding mode, immediate
fsrmi imm	csrrwi x0, frm, imm	Write FP rounding mode, immediate
frflags rd	csrrs rd, fflags, x0	Read FP exception flags
fsflags rd, rs	csrrw rd, fflags, rs	Swap FP exception flags
fsflags rs	csrrw x0, fflags, rs	Write FP exception flags
fsflagsi rd, imm	csrrwi rd, fflags, imm	Swap FP exception flags, immediate
fsflagsi imm	csrrwi x0, fflags, imm	Write FP exception flags, immediate

Pseudoinstruction	Base Instruction	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address
l{b|h|w|d} rd, symbol	auipc rd, symbol[31:12] l{b|h|w|d} rd, symbol[11:0](rd)	Load global
s{b|h|w|d} rd, symbol, rt	auipc rt, symbol[31:12] s{b|h|w|d} rd, symbol[11:0](rt)	Store global
fl{w|d} rd, symbol, rt	auipc rt, symbol[31:12] fl{w|d} rd, symbol[11:0](rt)	Floating-point load global
fs{w|d} rd, symbol, rt	auipc rt, symbol[31:12] fs{w|d} rd, symbol[11:0](rt)	Floating-point store global
nop	addi x0, x0, 0	No operation
li rd, immediate	*Myriad sequences*	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if ≠ zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero

fmv.s rd, rs	fsgnj.s rd, rs, rs	Copy single-precision register
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Single-precision absolute value
fneg.s rd, rs	fsgnjn.s rd, rs, rs	Single-precision negate
fmv.d rd, rs	fsgnj.d rd, rs, rs	Copy double-precision register
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Double-precision absolute value
fneg.d rd, rs	fsgnjn.d rd, rs, rs	Double-precision negate
beqz rs, offset	beq rs, x0, offset	Branch if $=$ zero
bnez rs, offset	bne rs, x0, offset	Branch if $\neq$ zero
blez rs, offset	bge x0, rs, offset	Branch if $\leqslant$ zero
bgez rs, offset	bge rs, x0, offset	Branch if $\geqslant$ zero
bltz rs, offset	blt rs, x0, offset	Branch if $<$ zero
bgtz rs, offset	blt x0, rs, offset	Branch if $>$ zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if $>$
ble rs, rt, offset	bge rt, rs, offset	Branch if $\leqslant$
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if $>$, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if $\leqslant$, unsigned
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, rs, 0	Jump register
jalr rs	jalr x1, rs, 0	Jump and link register
ret	jalr x0, x1, 0	Return from subroutine
call offset	auipc x6, offset[31:12] jalr x1, x6, offset[11:0]	Call far-away subroutine
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	Tail call far-away subroutine
fence	fence iorw, iorw	Fence on all memory and I/O

面对新兴指令集标准 RISC-V，很多人还在观望和比较。本书作者却“行胜于言”，完成了商业级水准的蜂鸟处理器。本书是你不得不读的 RISC-V 入门书籍和实践指南！

—— 景略半导体设计总监，RISC-V 爱好者和推广者　郭雄飞

这本书介绍了当今 CPU，尤其是 RISC-V 的最新技术和成果，还给出了设计实例（蜂鸟 E200）。内容上高屋建瓴，文采飞扬，深入浅出，是国内不可多得的理论联系实际、全面介绍 CPU 及芯片设计的好书，相信会对读者有所裨益！

—— 中科院“百人计划”海外引进杰出人才，物联网芯片设计公司创始人　胡国荣

这是国内较早关于 RISC-V 的著作，作者结合自己多年的处理器设计经验，以全新的视角分析介绍了 RISC-V 指令集和架构设计要点，是一本非常有价值的书！作者提供的开源 RISC-V 处理器设计和 SoC 平台，无论对于国内的高校、研究机构，还是相关开发的公司来说，都是难得的第一手资料。感谢作者为新一代处理器的推广和实践所做出的贡献，也希望本书读者能够体会到 RISC-V 处理器设计的精髓！

—— 上海交通大学微电子学院助理研究员　蒋剑飞

开源在软件世界普及之后，也开始进入硬件世界。RISC-V 开源精简指令集经过了几年的积累后，在 2017 年步入了快车道，获得了众多知名芯片公司和操作系统的支持。和 ARM 的过往一样，展望未来，RISC-V 将会成为芯片市场上一股不可忽视的力量。对于想了解此技术的开发者来说，本书绝对是首选。希望大家不要错过这本书，错过一个时代。

—— 半导体行业观察执行主编　李寿鹏

异步社区 www.epubit.com



异步社区 www.epubit.com
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn



ISBN 978-7-115-48052-1
定价：99.00 元

封面设计：广领设计
分类建议：计算机 / 硬件设计
人民邮电出版社网址：www.ptpress.com.cn